O'REILLY®



图灵程序设计丛书

JSON实战

JSON at Work

涵盖JSON基础知识、操作实践与案例

全面掌握JSON强大功能的明智之选

[美] 汤姆·马尔斯 著

邵钏 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

译者介绍

邵钏

毕业于浙江大学生物科学系，热衷于提升产品的用户体验，在UI技术领域历经Java Swing、Adobe Flex，终至Web前端。目前主要感兴趣的领域为物联网，并致力于相关产品Uniboard的设计开发。

数字版权声明



图灵程序设计丛书

JSON实战

JSON at Work

[美] 汤姆 · 马尔斯 著
邵钏 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（CIP）数据

JSON实战 / （美）汤姆·马尔斯（Tom Marrs）著 ;
邵钏译. -- 北京 : 人民邮电出版社, 2018.7
（图灵程序设计丛书）
ISBN 978-7-115-48555-7

Ⅰ. ①J… Ⅱ. ①汤… ②邵… Ⅲ. ①JAVA语言－程序
设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2018)第111633号

内 容 提 要

本书来自于作者实际使用 JSON 的经验所得，主要内容包括 JSON 基础知识，对 JSON 数据建模，在 Node.js、Ruby on Rails 和 Java 中使用 JSON，结构化 JSON 文档并设计测试 API，搜索 JSON 文档的内容，将 JSON 文档转换成其他数据格式，将 JSON 作为企业级架构中的一部分来使用，等等。

本书适合对 Web 和移动端应用、RESTful API 以及消息系统进行设计或实现的架构师和开发人员阅读。

◆ 著　　[美] 汤姆·马尔斯
译　　邵　钏
责任编辑　朱　巍
执行编辑　张海艳
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　印刷

◆ 开本：800×1000　1/16
印张：18.75
字数：443千字　　　　2018年 7 月第 1 版
印数：1－3 500册　　　　2018年 7 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-3441号

定价：89.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

献词

致在 Web/ 移动端应用程序、REST API 和消息系统中提供或使用 JSON 数据的所有人，希望本书能让你的工作变得更加简单。

致 JSON 社区中提供工具和类库的无名英雄，感谢你们为 JSON 所付出的艰辛劳动。

目录

第二部分　JSON 生态系统

第三部分　JSON 的企业级应用

前言

JavaScript 对象表示法（JavaScript Object Notation，JSON）已经成为 RESTful 接口设计中的事实标准，架构师和开发人员可以使用一整套现成的技术生态系统（鲜为人知的标准、工具和相关技术）来搭建设计精巧的应用程序。JSON 不仅仅是 Ajax 调用中 XML 的一个简单替代品，它也正日益成为互联网数据交换领域的骨干元素。严谨的标准和技术最佳实践加上对 JSON 的热爱，有助于我们搭建一个真正优雅、有用而又高效的应用程序。

唯一的缺憾是，没有一本书将这一切串连起来进行介绍。本书旨在帮助开发人员使用 JSON，以搭建企业级的应用程序与服务。我们的目标是促进 JSON 工具的使用，同时力图让**消息 / 文档设计**这一理念在日新月异的 API 社区中成为“一等公民”。

我和 JSON 的接触始于 2007 年，当时我正在负责一个大型的 Web 门户项目，而该项目要求实现拥有几千个选项的下拉列表。那时我刚好在阅读 Rebecca Riordan 所著的《Head First Ajax（中文版）》，因此设计了比较优雅的架构方案。Ajax 能够解决延迟和页面加载问题，但是该如何处理数据呢？前几年我一直在使用 XML 技术而且很成功，但对于将数据从 Web 应用程序后端传输到前端展现层这样的任务，继续使用 XML 技术显得有些大炮打蚊子。《Head First Ajax（中文版）》中提到了名为 JSON 的一种新数据格式，而这一策略看上去似乎是可行的。我的整个团队开始研究能将 Java 对象转换为 JSON 的 API，并最终选择了 JUnit 测试程序最简短的方案，我们的目标是在代码有效的前提下，尽可能简化所需的工作。我们对完成后的应用程序执行了严格的压力测试，而从 Java 转换为 JSON 的操作在测试中从未成为性能瓶颈。最终，这一应用程序在生产环境中呈现出了很好的可扩展性，用户也能瞬间看到下拉列表。

之后的一段时间里，我思考过在 Web 应用程序、RESTful API 和消息系统中均使用 JSON。2009 年，因为 XML Schema 可以在数据交换过程中提供语义校验，所以我仍旧在项目中使用 XML。当时我的技术决策是这样的：在 Web 用户界面上使用 JSON（出于速度考虑），在 Web Service 和消息系统中则使用 XML（出于数据集成考虑）。不过，当 2010 年听说 JSON Schema 后，我就意识到自己已经不再需要 XML 了。JSON Schema 标准目前还在完善中，但已经足够成熟，足以用于企业级应用程序中的数据集成任务。

时至今日，我已经习惯，或者更准确地说，迷上了 JSON。我开始在网络上搜索 JSON 的

其他功能，并发现了大量的 API、在线工具、内容搜索功能等。简而言之，能够使用 XML 实现的功能都可以（也应当）用 JSON 实现。

之后我开始搜索有关 JSON 的图书，却失望地发现只能在讲述 JavaScript 或者 RESTful Web Service 的书中，零星地找到一两章有关 JSON 的内容。JSON 社区欣欣向荣，拥有大量的支持工具、文章和博客，但除了 Douglas Crockford 的 JSON 官方网站，尚没有一处地方对这些知识和资源进行汇聚。

本书的目标读者

本书的目标读者是设计或实现 Web 和移动端应用程序、RESTful API 以及消息系统的架构师和开发人员。本书中的代码示例是用以下编程语言编写的：JavaScript、Node.js、Ruby on Rails 和 Java。如果你使用的是其他编程语言，如 Groovy、Go、Scala、Perl、Python、Clojure 或者 C#，同样需要阅读本书中的示例代码。不过你大可放心，绝大多数主流的现代编程语言都可以提供优秀的 JSON 支持。对于架构师，本书提供了指南、最佳实践以及架构和设计图表。然而，除了提供技术愿景，真正的架构师往往会用实际代码来佐证自己的观点。虽然我很喜欢编写代码来使用 JSON，但如果没有用例，缺少业务和技术背景，那么一切都将毫无意义。对于开发人员，本书汇聚了代码示例、工具、单元测试，等等。

为了保持简洁和专注，第 5~10 章仅提供在 Node.js 中编写的代码示例。但是，将这些示例转换为你使用的编程语言的代码并不难。

“实战”的含义

2000 年年中，当我与 Scott Davis 合作编写 *JBoss at Work* 时，我们的愿景是编写一本开发人员能在日常工作中使用的书。同样，本书的目的也是为开发人员提供实用示例，这些示例是我根据实际的 JSON 使用经验所编写的。为此，我在每章后面添加了单元测试（如果这一章的内容可以编写单元测试的话）。原因很简单：如果一段代码没有对应的测试，则该段代码不存在。

准备好卷起袖子看代码吧。无论你是架构师还是开发人员，本书都会对你的工作有所帮助。

本书内容

通过阅读本书并练习书中的示例，你将学到以下实战操作：

- JSON 基础知识，以及如何对 JSON 数据进行建模；
- 在 Node.js、Ruby on Rails 以及 Java 中使用 JSON；
- 使用 JSON Schema 结构化 JSON 文档来设计并测试 API；
- 使用 JSON 搜索工具来搜索 JSON 文档的内容；
- 使用 JSON 转换工具将 JSON 文档转换成其他数据格式；
- 将 JSON 作为企业级架构中的一部分来使用；
- 比较 HAL、`json:api` 等 JSON 超媒体格式；

- 使用 MongoDB 来存储和访问 JSON 文档；
- 使用 Apache Kafka 在服务间交换 JSON 消息；
- 使用免费的 JSON 工具来简化测试；
- 通过简单的工具和类库，使用自己偏好的编程语言来调用 API。

本书使用的工具

以下是本书中用到的 JSON 工具：

- JSON 编辑器 / 建模工具；
- 单元测试工具（如 Mocha/Chai、Minitest、JUnit）；
- JSON 校验工具；
- JSON Schema 生成器；
- JSON 搜索工具；
- JSON 转换（模板）工具。

本书不适合哪些读者

如果对 JSON 的兴趣仅限于用 JavaScript 来发起 Ajax 调用，那么本书并不适合你。虽然本书也涉及了 Ajax 调用，但这只是所有内容中的冰山一角。有关 JavaScript 的很多图书中都包含有关 Ajax 调用的章节。

本书不会包含 REST、Ruby on Rails、Java 和 JavaScript 等内容的深入介绍。本书会用到上述技术，但将关注点放在了如何通过这些技术来使用 JSON 上。

本书的架构

本书由以下几部分内容组成：

- 第一部分，JSON 概述与平台；
- 第二部分，JSON 生态系统；
- 第三部分，JSON 的企业级应用；
- 附录。

第一部分，JSON概述与平台

- **第 1 章　JSON 概述**
  这一章从概述 JSON 数据格式开始，描述使用 JSON 过程中的最佳实践，并介绍本书中所使用的工具。

- **第 2 章　在 JavaScript 中使用 JSON**
  这一章展示了如何在 JavaScript、Node.js、Mocha/Chai 单元测试中使用 JSON。

- **第 3 章　在 Ruby on Rails 中使用 JSON**
  这一章描述了如何在 Ruby 对象和 JSON 之间进行转换，以及如何与 Rails 进行集成。

- **第 4 章　在 Java 中使用 JSON**
  这一章讲述了如何在 Java 和 Spring Boot 中使用 JSON。

第二部分，JSON生态系统

- **第 5 章　JSON Schema**
  这一章将帮助你用 JSON Schema 对 JSON 文档进行结构化操作。同时，你还会学习如何生成 JSON Schema 并用其来设计 API。
- **第 6 章　在 JSON 中进行搜索**
  这一章展示了如何通过 `jq` 和 JSONPath 搜索 JSON 文档。
- **第 7 章　JSON 转换**
  这一章提供了工具，从而将设计糟糕的 JSON 文档转换为更优雅、更有用的 JSON 文档。这一章还介绍了如何在 JSON 与 XML、HTML 等其他格式间进行相互转换。

第三部分，JSON的企业级应用

- **第 8 章　JSON 与超媒体**
  这一章介绍了如何在 JSON 中使用多种知名的超媒体格式，如 HAL 和 `jsonapi`。
- **第 9 章　JSON 与 MongoDB**
  这一章展示了如何使用 MongoDB 来存储与处理 JSON 文档。
- **第 10 章　用 Kafka 实现 JSON 消息系统**
  这一章描述了如何使用 Apache Kafka 在服务间交换 JSON 消息。

附录

- 附录 A 介绍了如何安装运行本书示例所需的应用程序。
- 附录 B 提供了更多有关 JSON 社区（如标准、教程）的信息与链接，有助于你深入学习 JSON。

代码示例

本书中的所有代码示例及网址链接均可在图灵社区本书页面免费下载：http://www.ituring.com.cn/book/2093。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*JSON at Work* by Tom Marrs (O'Reilly). Copyright 2017 Vertical Slice, Inc., 978-1-449-35832-7."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O’Reilly Safari

Safari（前身为 Safari Books Online，http://oreilly.com/safari）是一个会员制的培训和参考平台，面向企业、政府、教育从业者和个人。



Safari 用户可访问 O’Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等 250 多家出版社的上千种图书、培训视频、学习路径、交互式教程和精选播放列表。

如需了解更多信息，请访问 http://oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：http://shop.oreilly.com/product/0636920028482.do。

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

首先，我要感谢创建并标准化 JSON 数据格式的 Douglas Crockford。JSON 是 REST 和微服务领域数据所用的语言，整个社区都应该感谢 Crockford 的愿景与付出。

感谢 O'Reilly 本书的编辑 Megan Foley 和前任编辑 Simon St. Laurent，感谢他们对本书的信任，以及在本书出版过程中给予我的耐心与指导。感谢他们一直支持我，并帮助我完成整个项目。还要感谢这本书的文字编辑和制作编辑 Nick Adams 和 Sharon Wilkey，感谢他们孜孜不倦地改进本书质量。

感谢 O'Reilly 开源大会的 Matthew McCullough 和 Rachel Roumeliotis，No Fluff Just Stuff（NFJS）大会的 Jay Zimmerman，以及 Great Indian Developer Summit（GIDS）大会的 Dilip Thomas，感谢他们让我在会议上分享有关 JSON 和 REST 的内容。在会议上进行分享非常有趣，希望未来还能有这样的机会。

感谢对本书提出宝贵反馈的技术审稿人：Joe McIntyre、David Bock、Greg Ostravich 和 Zettie Chinfong。此外，还要感谢以下人员，他们帮助我梳理了讲述 JSON 的思路：Matthew McCullough、Scott Davis、Cristian Vyhmeister、Senthil Kumar、Sean Pettersen、John Gray、Doug Clark、Will Daniels、Dan Carda 和 Peter Piper。

科罗拉多的技术社区是世界一流的，而我也有幸在以下用户组中做过分享，这对改进本书材料不无帮助：

- HTML5 Denver
- Denver Open Source User Group（DOSUG）
- Colorado Springs Open Source User Group（CS OSUG）
- Denver Java User Group（DJUG）
- Boulder Java User Group（BJUG）
- BoulderJS Meetup

感谢鼓励、信任和推动我出版本书的 Toastmasters 社区中的朋友：Darryle Brown、Deborah Frauenfelder、Eli-nora Reynolds、Betty Funderburke、Tom Hobbs、Marcy Brock，以及其他很多很多人。他们启发我要清晰地沟通、帮助同行、放开眼界。

互联网上的 JSON 社区非常棒。本书中的很多内容都是对社区优秀工作成果的阐述。技术社区启发我将 JSON 知识点串连起来，并讲述整个 JSON 技术图景。

感谢我已故的父母 Al Marrs 和 Dorene Marrs 一直以来对我的爱、信任和支持，我知道你们已经在一个更美好的地方了。你们启发我学会适应、创新并努力工作。你们总是鼓励我全力以赴。感谢你们为我做的一切。

最后，感谢我美丽的妻子 Linda 和女儿 Abby——我爱你们。当我将晚上和周末的时间用于写作和编程时，谢谢你们对我的耐心和理解。

电子书

扫描如下二维码，即可购买本书电子版。



第一部分

JSON概述与平台

第 1 章

JSON概述

JSON 数据格式使得应用程序可以通过 RESTful API 等方式在网络上进行数据通信。JSON 不局限于某项技术，本身非私有，且可移植。对于产生（序列化）和读取（反序列化）JSON 数据，所有的现代编程语言（Java、JavaScript、Ruby、C#、PHP、Python、Groovy 等）和平台都提供了良好的支持。JSON 非常简单，由对象、数组和名称－值对这三种开发人员所熟悉的结构体所组成。除了**表现层状态转化**（REpresentational State Transfer，REST），JSON 在以下环境中也有所应用：

- Node.js（在 package.json 中存储项目元数据）；
- MongoDB 等 NoSQL 数据库（详见第 9 章）；
- Kafka 等消息平台（详见第 10 章）。

1.1　JSON是一项技术标准

早年间，REST 的反对者在批评 RESTful Web Service 时曾认为它并不是标准，但与 HTTP 一样，JSON 确实是一项技术标准。无论是互联网工程任务组（Internet Engineering Task Force，IETF），还是 Ecma 国际（前身为欧洲计算机制造商协会，European Computer Manufacturers Association，ECMA），都已经认可这一点。JSON 由 Douglas Crockford 于 2001 年提出，并在 2006 年由 IETF 通过 RFC 4627 进行首次标准化。2013 年秋，Ecma 国际通过 ECMA 404 将 JSON 正式标准化。得到 Ecma 国际的承认后，Douglas Crockford 认为 JSON 已经是一个正式的国际数据处理标准了。

2014 年 3 月，Tim Bray 发布了 RFC 7158 和 RFC 7159，以作为 Douglas Crockford 原始标准的改进版。这两份文档修正了之前 RFC 4627 标准中的一些错误，并将其状态更改为“废弃”。

1.2 示例

在继续深入前，我们先查看 JSON 的一个小示例。例 1-1 展示了一个简单的 JSON 文档。

例 1-1 firstValidObject.json

```
{ "thisIs": "My first JSON document" }
```

一个合法的 JSON 文档一般属于以下两种情况之一：

- 由大括号 { 和 } 括起来的一个对象；
- 由中括号 [和] 括起来的一个数组。

例 1-1 展示了一个包含单个名称 – 值对的对象，其中键 `"thisIs"` 的值为 `"My first JSON document"`。

为了证明这个 JSON 文档是合法的，我们使用 JSONLint 来校验一下。将上述 JSON 文本粘贴到文本输入框中，然后点击 Validate 按钮，就会看到如图 1-1 所示的页面。

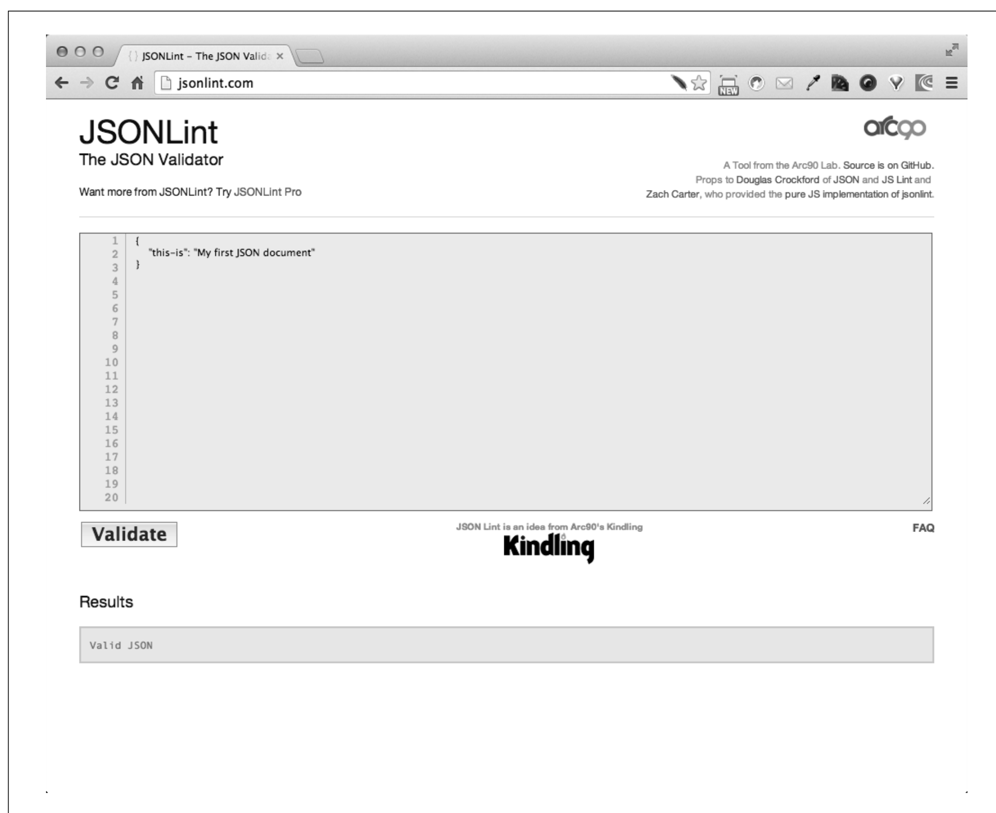


图 1-1：一个简单、合法的 JSON 文档在 JSONLint 中的测试结果

例 1-2 展示了一个简单的 JSON 数组。

例 1-2　firstValidArray.json

```
[
  "also",
  "a",
  "valid",
  "JSON",
  "doc"
]
```

在 JSONLint 中，将 JSON 数组粘贴到文本输入框中，然后点击 Validate 按钮，就会看到如图 1-2 所示的结果。

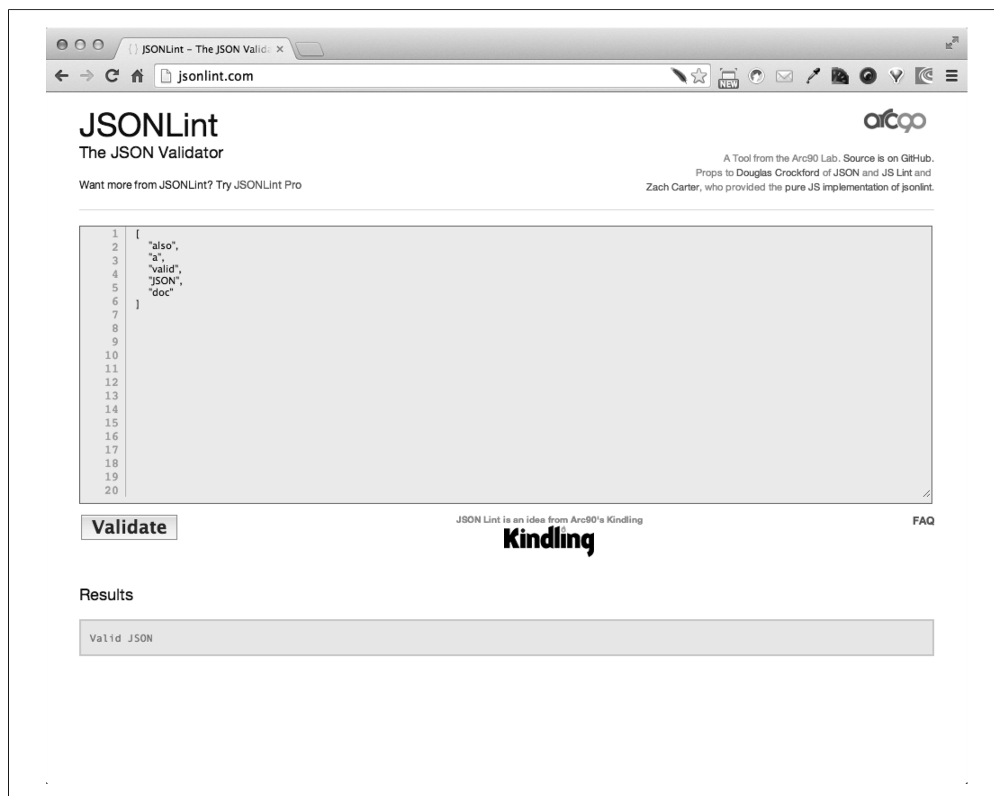


图 1-2：合法的 JSON 数组在 JSONLint 中的测试结果

我们已经有些超前了。1.4 节会深入阐述 JSON 的语法。

1.3　为什么使用JSON

Ecma 国际和 IETF 所做的标准化工作帮助 JSON 获得了行业认可，但使 JSON 广为流行的却是其他一些因素：

- 基于 JSON 的 RESTful API 的爆发式增长；
- JSON 基本数据结构的简洁性；
- JavaScript 日渐流行。

JavaScript 的复兴推动了 JSON 的流行。在过去几年中，我们见证了 JavaScript 作为一门顶级编程语言的崛起。JavaScript 生态系统既包含了 Node.js 这样的平台，也包含了 AngularJS、React、Backbone 和 Ember 这样的模型 / 视图 / 控制器（Mode/View/Controller，MVC）框架。有关 JavaScript 对象和模式最佳实践的图书和网站也层出不穷。按照 Douglas Crockford 的说法，JSON 是 JavaScript 对象字面量表示法的一个子集，因此可以无缝地与 JavaScript 开发融为一体。

数以千计的 RESTful API 使用了 JSON。以下是基于 JSON 的一些流行的 RESTful API：

- LinkedIn
- Twitter
- Facebook
- Salesforce
- GitHub
- DropBox
- Tumblr
- Amazon Web Services（AWS）

如果想要查看这几千个基于 JSON 的 RESTful API，可以访问 ProgrammableWeb，搜索关键词 REST 和 JSON，然后花上好几周来查阅结果。

JSON 非常简洁，并且正在逐步替代 XML 成为互联网上主要的数据交换格式。它易于阅读，相关结构也很容易与软件开发人员所熟悉的概念对应起来，如数组、对象和名称 - 值对。我们不用再挠头苦思某个东西应当是元素还是属性，也不用再就这一点与人争论不休。与 XML 相比，对象及其数据成员这一组合更适合面向对象的设计和开发。由于节省了每个数据元素的开始标签与结束标签，JSON 格式的额外开销更少、更为紧凑，所以 JSON 格式的文档一般比内容相同的 XML 文档小。从企业级应用的角度来看，与 XML 相比，JSON 文档在网络上的传输与处理更快，因此效率更高。

虽然 Douglas Crockford 在提出 JSON 时将其设计为一种数据交换格式（通常用于 REST），但如今 JSON 在配置文件领域也占有一席之地，如 Node.js 和 Sublime Text 等广泛使用的产品。Node.js 使用 package.json 文件来定义其标准的 npm 包结构，第 2 章将对此进行详细阐述。Sublime Text 则是 Web 开发社区中流行的一款 IDE，它使用 JSON 来配置外观及包管理器。

1.4　JSON的核心概念

JSON 的核心数据格式包括 JSON 数据类型和 JSON 值类型。对于 JSON 的版本、注释以及文件 /MIME 类型，本节也会有所提及。

1.4.1　JSON数据类型

JSON 包括以下 3 种核心数据类型。

名称 – 值对
: 由一个名称（数据属性）和一个值组成。

对象
: 名称 – 值对的无序集合。

数组
: 值的有序集合。

描述了基本定义后，我们来深入了解一下每种数据类型。

1. 名称 – 值对

例 1-3 展示了名称 – 值对的一个示例。

例 1-3　nameValue.json

```
{
  "conference": "OSCON",
  "speechTitle": "JSON at Work",
  "track": "Web APIs"
}
```

名称 – 值对具有以下特征。

- 每一个键名（如 "conference"）
  - 位于冒号（:）左边；
  - 是一个字符串，而且必须由双引号括起来。
- 值（如 "OSCON"）位于冒号的右边。在上述示例中，值的类型为字符串，但还存在多种其他值类型。

1.4.2 节将对字符串和其他合法的值类型进行介绍。

2. 对象

对象由名称 – 值对组成。例 1-4 展示了一个表示地址的对象。

例 1-4　simpleJsonObject.json

```
{
  "address" : {
    "line1" : "555 Any Street",
    "city" : "Denver",
    "stateOrProvince" : "CO",
    "zipOrPostalCode" : "80202",
    "country" : "USA"
  }
}
```

例 1-5 展示了一个带有内嵌数组的对象。

例 1-5　jsonObjectNestedArray.json

```
{
  "speaker" : {
    "firstName": "Larson",
    "lastName": "Richard",
    "topics": [ "JSON", "REST", "SOA" ]
  }
}
```

例 1-6 展示了一个内嵌其他对象的对象。

例 1-6　jsonObjectNestedObject.json

```
{
  "speaker" : {
    "firstName": "Larson",
    "lastName": "Richard",
    "topics": [ "JSON", "REST", "SOA" ],
    "address" : {
      "line1" : "555 Any Street",
      "city" : "Denver",
      "stateOrProvince" : "CO",
      "zipOrPostalCode" : "80202",
      "country" : "USA"
    }
  }
}
```

对象具有以下特征：

- 由左大括号 { 和右大括号 } 括起来；
- 由一些无序的名称 – 值对组成，以逗号分隔；
- 可以是空对象 { }；
- 可以内嵌在其他对象或者数组中。

3. 数组

例 1-7 展示了一个内嵌其他对象和数组的数组，该数组描述了包含标题、长度和摘要信息的会议报告。

例 1-7　jsonArray.json

```
{
  "presentations": [
    {
      "title": "JSON at Work: Overview and Ecosystem",
      "length": "90 minutes",
      "abstract": [ "JSON is more than just a simple replacement for XML when",
                    "you make an AJAX call."
                  ],
      "track": "Web APIs"
    },
    {
      "title": "RESTful Security at Work",
      "length": "90 minutes",
      "abstract": [ "You've been working with RESTful Web Services for a few years",
```

```
                    "now, and you'd like to know if your services are secure."
                  ],
          "track": "Web APIs"
        }
      ]
    }
```

数组具有以下特征：

- 由左中括号 [和右中括号] 括起来；
- 由一些有序的值组成，以逗号分隔（详见下节）；
- 可以是空数组 []；
- 可以内嵌在其他数组或者对象中；
- 具有以 0 或 1 开头的索引。

1.4.2　JSON值类型

JSON 值类型用于表示出现在名称－值对冒号（:）右侧的数据类型。这些类型包括：

- 对象
- 数组
- 字符串
- 数值
- 布尔值
- null

前面已经介绍过对象和数组，接下来我们看一下其他的值类型：字符串、数值、布尔值和 null。

1. 字符串

例 1-8 展示了一些合法的 JSON 字符串。

例 1-8　jsonString.json

```
[
  "fred",
  "fred\t",
  "\b",
  "",
  "\t",
  "\u004A"
]
```

字符串具有以下特征。

- 由包含在双引号（""）间的零个或多个 Unicode 字符组成。除此之外，还包括以下列举的一些字符。
- 由单引号（'）引起来的字符串为非法字符串。

JSON 字符串还可以包含由反斜杠转义的字符，如下所示：

`\"`
: 双引号

`\\`
: 反斜杠

`\/`
: 正斜杠

`\b`
: 退格

`\f`
: 换页

`\n`
: 换行

`\r`
: 回车

`\t`
: Tab 制表符

`\u`
: 后跟 4 个十六进制数字（表示一个 Unicode 字符）

2. 数值

例 1-9 展示了一些合法的 JSON 数值。

例 1-9　jsonNumbers.json

```
{
  "age": 29,
  "cost": 299.99,
  "temperature": -10.5,
  "unitCost": 0.2,
  "speedOfLight": 1.23e11,
  "speedOfLight2": 1.23e+11,
  "avogadro": 6.023E23,
  "avogadro2": 6.023E+23,
  "oneHundredth": 10e-3,
  "oneTenth": 10E-2
}
```

数值遵循 JavaScript 的双精度浮点数格式，并且具有以下特征。

- 数值永远是十进制数（只能出现数字 0~9），不能以 0 开头。
- 数值可以存在由小数点（.）开头的小数部分。
- 数值可以是以 10 为底的指数，该指数由 `e` 或 `E` 来表示，其后跟正号表示正指数幂，跟负号则表示负指数幂。
- 数值不支持八进制数和十六进制数。

- 与 JavaScript 不同，数值不能是 NaN（Not a Number，用于表示非法数值），也不能是 Infinity。

3. 布尔值

例 1-10 展示了 JSON 中的布尔值。

例 1-10　jsonBoolean.json

```
{
  "isRegistered": true,
  "emailValidated": false
}
```

布尔值具有以下特征。

- 只存在两种值：`true` 或 `false`。
- 冒号（:）右边的 `true` 或者 `false` 不能由引号括起来。

4. `null`

从技术上来说，`null` 并不是一种值类型，而是 JSON 中的一个特殊值。例 1-11 展示了 `line2` 这一属性的值为 `null`。

例 1-11　jsonNull.json

```
{
  "address": {
    "line1": "555 Any Street",
    "line2": null,
     "city": "Denver",
        "stateOrProvince": "CO",
        "zipOrPostalCode": "80202",
        "country": "USA"
    }
}
```

`null` 具有以下特征。

- 不能由引号括起来。
- 表示某个键或属性没有值。
- 用作占位符。

1.4.3　JSON的版本

根据 Douglas Crockford 的说法，JSON 的核心标准不会再有新的版本。这倒不是说 JSON 是完美的，毕竟没有什么是完美的。JSON 标准唯一化的目的是避免为支持早期版本而在向后兼容的过程中遭遇陷阱。Crockford 认为，当开发社区有新的需要时，新的数据格式将替代 JSON。

但正如接下来的章节所描述的，“无版本”的理念仅适用于 JSON 的核心数据格式。比如，第 5 章中提到的相关标准在编写本书时的版本号为 0.5。值得一提的是，与 JSON 有关的这些标准是由 JSON 社区中的其他人员所提出的。

1.4.4　JSON中的注释

一言以蔽之，JSON 中没有注释。

根据 Crockford 在 Yahoo! JSON group 和 Google+ 上的说法，JSON 最开始是允许出现注释的，但之后不久就因为以下原因移除了注释。

- Crockford 认为注释没有什么用处。
- JSON 解析器在支持注释方面存在困难。
- 出现了滥用注释的情况。Crockford 发现有些注释被用于解析指令，而这会彻底摧毁 JSON 的互操作性。
- 移除注释有利于 JSON 实现跨平台性，简化这方面的支持工作。

1.4.5　JSON文件及MIME类型

根据 JSON 的核心标准文档，.json 是在文件系统中存储 JSON 数据的标准文件类型。在互联网号码分配局（Internet Assigned Numbers Authority，IANA）中，JSON 的媒体类型（或者说 MIME）为 `application/json`，这可以在 IANA 的媒体类型网站上找到。通过在 HTTP 头部中声明 JSON 媒体类型，RESTful Web Service 的开发者和使用者一般使用这种名为**内容协商**的机制来表明自己正在使用 JSON 进行数据交换。

1.4.6　JSON编码规范

JSON 的意义在于互操作性，因此以使用者期望的方式提供数据是非常重要的。为了总结最佳实践，提升可维护性，Google 发布了 **JSON 编码规范**。

这份规范非常详细，其中对于 API 设计者和开发者来说最重要的是以下三点：

- 属性名；
- 日期属性的值；
- 枚举值。

1. 属性名

按照 Google 的说法，属性名位于名称 – 值对中冒号的左侧（属性值则位于右侧）。JSON 属性名的风格主要有两种：

- 小驼峰式命名法（`lowerCamelCase`）；
- 以下划线分隔的短语（`snake_case`）。

当使用小驼峰式命名法时，属性名是由单个或多个词拼接而成的一个“单词”，其中除第一个词外，其余每个词均以大写字母开头。Java 社区和 JavaScript 社区在编码规范上均使用了小驼峰式命名法。当使用以下划线分隔的短语时，所有的字母均小写，词与词之间以下划线（_）分隔。Ruby on Rails 社区更偏好这种命名法。

与多数 RESTful API 一样，Google 在属性名上使用了小驼峰式命名法，如例 1-12 所示。

例 1-12　jsonPropertyName.json

```
{
  "firstName": "John Smith"
}
```

2. 日期属性的值

你可能认为日期的格式并不那么重要，但实际上其重要性毋庸置疑。想象在来自不同国家或大陆的数据提供者与使用者之间交换日期信息。即使在同一家公司，两个开发小组也可能使用不同的日期格式。思考在语义上如何解释时间戳，从而在所有时区采用一致的日期 / 时间处理机制并保持互操作性，是非常重要的。Google 的 JSON 编码规范偏好让日期遵循 RFC 3339 的格式，如例 1-13 所示。

例 1-13　jsonDateFormat.json

```
{
  "dateRegistered": "2014-03-01T23:46:11-05:00"
}
```

以上日期使用了协调世界时（Coordinated Universal Time，UTC）偏移 −5 小时的时区（以格林尼治标准时间 UTC/GMT 作为基准计算），即美国东部时间。值得注意的是，RFC 3339 是 ISO 8601 的一个概括。两者的主要区别在于，ISO 8601 允许将用于分隔日期和时间的 `T` 字符替换为空格，RFC 3339 则不允许这么做。

3. 经纬度值

Google Maps 等地理信息 API 以及与地理信息系统相关的其他 API 会用到经纬度数据。为了保持一致性，Google 的 JSON 编码规范建议使用经纬度数据时遵循 ISO 6709 标准。根据 Google Maps 所提供的信息，美国纽约帝国大厦的经纬度为北纬 40.748747 度，西经 73.985547 度，这在 JSON 中的表示如例 1-14 所示。

例 1-14　jsonLatLon.json

```
{
  "empireStateBuilding": "40.748747-73.985547"
}
```

上例中的经纬度以 `±DD.DDDD±DDD.DDDD` 的格式表示，同时遵循以下约定。

- 纬度在前，经度在后。
- 北半球的纬度为正数。
- 本初子午线以东的经度为正数。
- 经纬度值以字符串表示。因为可能存在负号，所以无法用数值表示作为一个整体的经纬度。

4. 缩进

虽然 Google 的 JSON 编码规范并未提及缩进这一话题，不过大致存在以下规则。

- JSON 是一种序列化格式，而不是呈现格式。因此，缩进对于 API 的提供者和使用者来说意义不大。
- 在优化 JSON 文档的显示时，很多 JSON 格式化工具会让使用者选择具体的缩进方案（两格缩进、三格缩进、四格缩进等）。

- JSON 起源于 JavaScript，并已成为 ECMA 262 标准的一部分。但遗憾的是，JavaScript 社区在缩进问题上并未达成共识。很多人以及不少编码规范中都偏好使用两格缩进，因此本书也遵循了这一惯例。当然，只要保持一致，你也可以选择其他缩进风格。

1.5 本书示例：MyConference

本书的所有示例都与会议数据有关，具体包括以下两方面：

- 会议的演讲者；
- 会议的主题演讲。

1.5.1 本书技术栈

我们将先创建一段简单的 JSON 数据，用于表示会议演讲者，然后将其发布为一个模拟的 RESTful API。具体的操作步骤如下所示。

(1) 用 JSON Editor Online 对 JSON 数据进行建模。
(2) 用 JSON Generator 生成示例数据。
(3) 创建并部署模拟 API，为之后的测试工作做准备。

1.5.2 本书架构风格：noBackend

本书的架构风格是基于 noBackend 的理念形成的。如果采用 noBackend 的理念，开发人员就无须在应用程序开发的早期阶段，烦心于服务器和数据库等具体细节。

本书前 7 章的示例采用了 noBackend 的架构，目的是从业务角度（服务与数据优先）关注应用程序的开发，从而使得应用程序不仅能支持移动端、平板和网页等 UI 客户端，同时也能支持 API 调用和非 Web 客户端。JSON 数据会由 `json-server` 这样的小工具来部署，从而模拟实现 RESTful API。

采用这一接口先行的策略来设计和构建 API 能带来诸多好处。

- 因为与后端解耦，所以前端开发会变得更加敏捷、快速和迭代化。
- API 本身能够更加快速地获得反馈。将 API 的 URI 和数据快速推向市场有助于更快得到审阅。
- 有助于得到一个更加清晰、整洁的 API 接口。
- 关注点分离。将 API 暴露的资源（如以 JSON 数据表示的会议演讲者）与其最终内部实现（如应用程序服务器、业务逻辑、数据存储等）分离。这可以使得后续修改具体实现变得更加简单。如果过早创建和部署由 Node.js、Rails、Java（或者其他框架）所开发的真实 API，则意味着在非常早的阶段就确定了具体的设计决策，这会导致最后和 API 的使用者进行磨合时，修改的难度会增加。

使用模拟 API 能够带来以下便利。

- 不必在项目初期进行服务器和数据库方面的工作。
- 允许 API 的提供者（写 API 的开发人员）专注于 API 设计，聚焦于如何用最好的方式向使用者呈现数据，同时执行最初阶段的测试。

- 使得 API 的使用者（UI 开发人员）能够在早期阶段和 API 对接，并向 API 的开发团队提供反馈。

在编写代码并部署到服务器之前，通过使用本书中所提到的轻量级工具，你可以完成很多工作。当然，最终还是需要实现 API 的，第 2~4 章在介绍 JavaScript、Ruby on Rails 和 Java 时会展示具体的实现方法。

1.5.3 用JSON Editor Online对JSON数据进行建模

创建大小和复杂度都比较真实的合法 JSON 文档繁琐且易错。在这一点上，JSON Editor Online 是一个很不错的 Web 工具，提供了以下便利。

- 支持用对象、数组和名称 – 值对的方式对 JSON 文档进行建模。
- 使得快速遍历生成 JSON 文档的操作变得更加简单。

JSONmate 是另一个比较不错的编辑器，但本书不会详细介绍。

1. JSON Editor Online的功能

除了 JSON 建模和文档生成，JSON Editor Online 还提供以下功能。

JSON 校验

　当在页面左侧的 JSON 文本框中输入 JSON 数据时，这些数据就会被校验。如果遗漏了某个值的闭合双引号（如 `"firstName":"Ester,`），则 JSON 文本中的下一行会出现 `X` 标记。当鼠标悬浮在该标记上时，就会显示校验错误的具体原因。

优化显示效果

　点击 JSON 文本框区域左上角的 Indent 按钮可以优化 JSON 文档的显示效果。

模型与文本之间的双向工程

　在页面右侧的 JSON 模型中使用添加按钮（`+`），可以往模型中添加一些对象和名称 – 值对，然后点击页面中间偏上方的左箭头按钮即可生成 JSON 文本。可以在页面左侧的 JSON 文本框中看到相应的变化。

同样，在 JSON 文本框中对数据进行修改，然后点击右箭头按钮即可在页面右侧的 JSON 模型中看到相应的改变。

将 JSON 文档保存到磁盘

　通过选择 Save 菜单中的 Save to Disk 选项，可以将 JSON 文档保存到本地机器中。

导入 JSON 文档

　通过选择 Open 菜单中的 Open from Disk 选项，可以导入计算机中的 JSON 文档。

需要注意的是，JSON Editor Online 是一个公用工具，粘贴到该应用程序中的所有数据对于其他人来说都是可见的。因此，请勿使用该工具处理个人、财产等隐私信息。

2. JSON Editor Online中的会议演讲者数据

完成对演讲者数据的建模后，点击左箭头按钮，即可生成优化缩进显示后的 JSON 文档。图 1-3 展示了 JSON Editor Online 以及初始演讲者数据模型。

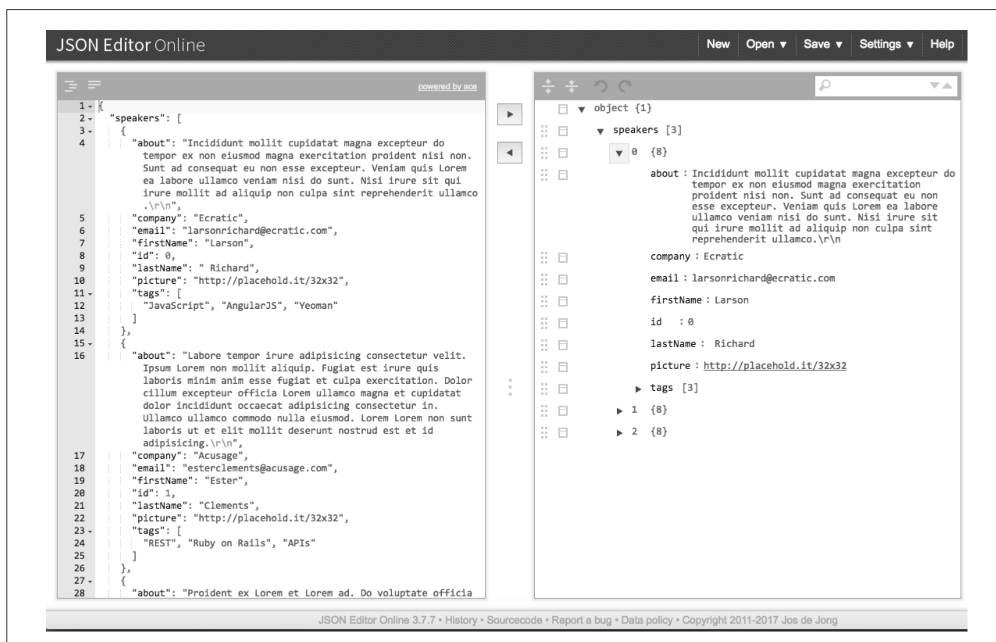


图 1-3：JSON Editor Online 中的演讲者数据模型

尽管这个模型比较粗糙，却是一个不错的开始。可以使用这一模型对 JSON 数据进行可视化、在早期阶段收集反馈，并快速对设计进行迭代。这一策略可以帮助你在整个开发周期中改善 JSON 数据结构，而无须在基础架构和具体实现上耗费大量时间和精力。

1.5.4　用JSON Generator生成示例数据

使用 JSON Editor Online 是一个不错的开端，但我们真正需要的其实是快速生成**大量的**测试数据。由于隐私数据的敏感性，以及任何正式测试对数据量的庞大需求，生成测试数据会显得比较棘手。对于这一问题，即使使用 JSON Editor Online，也会在创建所需的测试数据上耗费大量精力。因此，我们需要别的工具来帮助生成第一版 API 所需的数据。这个优秀工具就是用于生成测试数据文件 speakers.json 的 JSON Generator。用于生成 speakers.json 文件的模板可以在 GitHub 上找到。第 5 章将详细介绍 JSON Generator 的用法。

1.5.5　创建并部署模拟API

我们会将刚创建的会议演讲者测试数据部署为 RESTful API，从而创建出一个模拟的 API 服务。为了将 speakers.json 文件暴露为 Web API，从而快速制作原型，我们会使用 `json-server` 这一 Node.js 模块。关于 `json-server`，可以在其 GitHub 页面上找到更多相关信息。

在继续深入之前，需要搭建好开发环境，请先参考附录 A 来执行以下步骤。

(1) 安装 Node.js。`json-server` 是一个 Node.js 模块，因此首先得安装 Node.js。可以参考 A.2 节中的相关内容。

(2) 安装 `json-server`，可以参考 A.2.5 节。

(3) 安装 JSONView 和 Postman，可以参考 A.1 节。JSONView 可以在 Chrome 和 Firefox 中优化 JSON 的显示。Postman 可以在大多数主流操作系统中以独立的 GUI 应用程序的形式来运行。

打开终端，在命令行中用 5000 端口来启动 `json-server`，如下所示：

```
cd chapter-1

json-server -p 5000 ./speakers.json
```

可以看到以下执行结果：

```
json-at-work => json-server -p 5000 ./speakers.json

  \{^_^}/ hi!

  Loading ./speakers.json
  Done

  Resources
  http://localhost:5000/speakers

  Home
  http://localhost:5000

  Type s + enter at any time to create a snapshot of the database
```

在浏览器中访问 http://localhost:5000/speakers，即可看到由模拟 API 所提供的（经过 JSONView 优化显示的）所有演讲者数据，如图 1-4 所示。

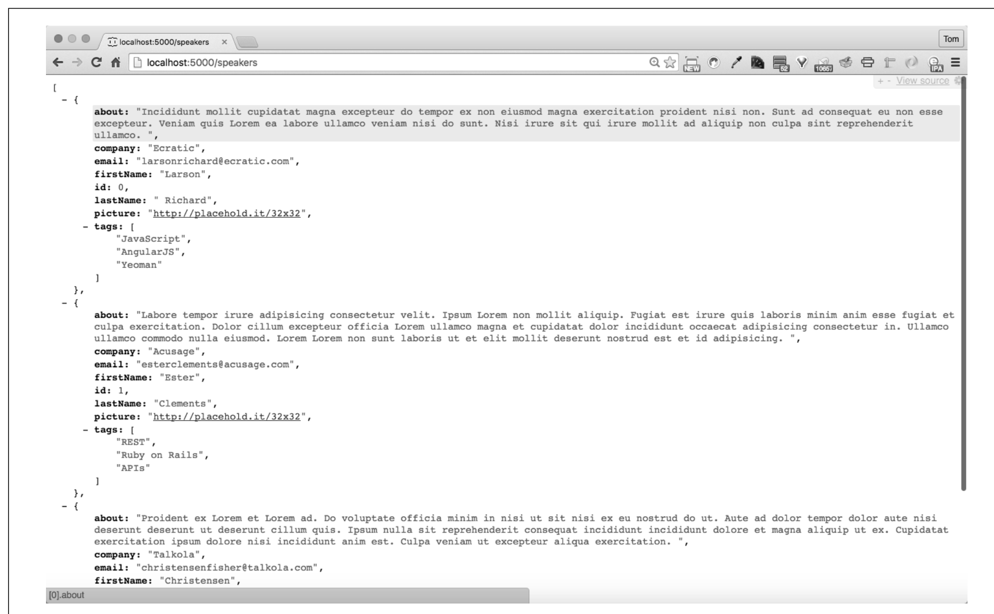


图 1-4：在浏览器中用 JSONView 显示由 `json-server` 提供的演讲者数据

还可以通过往 URI 中添加 `id` 来获取单个演讲者的信息，如 http://localhost:5000/speakers/0。

效果还不错，但使用浏览器进行测试时只能发送 HTTP `GET` 请求，因此存在一定的局限性。而 Postman 既可以发送 HTTP 的 `GET`、`POST`、`PUT` 和 `DELETE` 请求，还可以设置 HTTP 头部，因此具备对 RESTful API 进行完整测试的能力。

可以使用 Postman 删除 API 中第一个演讲者的数据，具体步骤如下所示。

(1) 输入 URL：http://localhost:5000/speakers/0。
(2) 选择 `DELETE` 作为 HTTP 方法。
(3) 点击 Send 按钮。

Postman 中的 `DELETE` 请求正常运行，获取到的 HTTP 响应码为 200（OK），如图 1-5 所示。

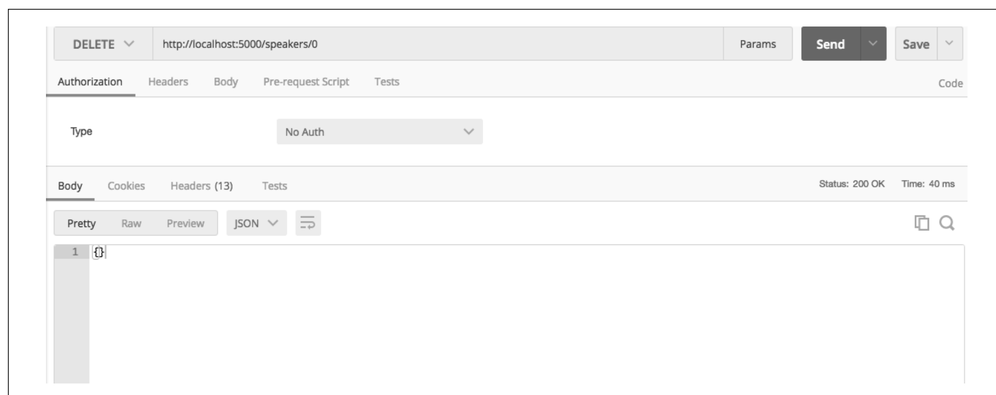


图 1-5：Postman 删除第一个演讲者的请求结果

在浏览器中再次访问 http://localhost:5000/speakers/0，以确认删除了第一个演讲者的数据。响应的数据应当是空的，如图 1-6 所示。

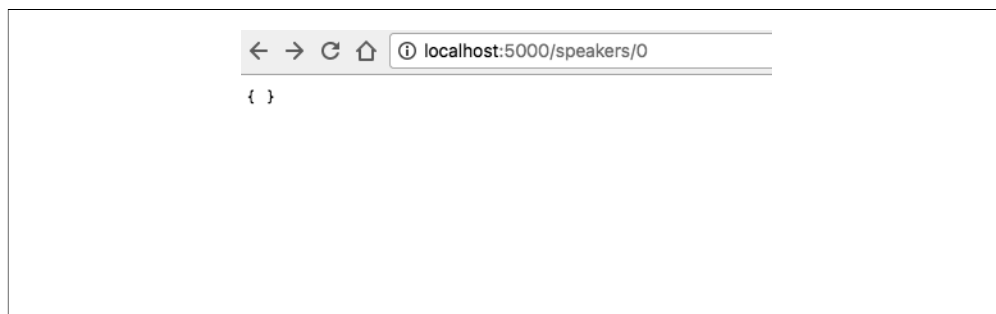


图 1-6：验证删除第一个演讲者的结果

如果需要关闭 `json-server`，可在命令行中敲击 Ctrl-C。

有了模拟 API 后，就可以用任意一种 HTTP 客户端（JavaScript、Ruby 或者 Java 等）对其进行调用，以外部应用程序的形式使用测试数据。虽然本书后续章节中的示例大多使用了

HTTP `GET` 方法，但实际上 `json-server` 支持所有核心的 HTTP 请求类型（`GET`、`POST`、`PUT`、`DELETE`）。值得一提的是，本书中并未详细介绍的 Mountebank 也是一个不错的服务器工具，在对 API 和协议的模拟方面可以提供更健壮的功能。

使用模拟 API 的关键点在于：在不编写任何代码的情况下，API 的提供者可以利用 JSON 工具来制作可测试的 RESTful API 原型。这一技术使得 API 的使用者无须等待 API 100% 完成即可开始测试工作，因此非常有用。在 API 的使用者测试模拟 API 的同时，API 的开发团队可以对原型及其设计进行迭代式的改进。

1.6 本章回顾

本章先介绍了 JSON 的基础知识，然后用 JSON Editor Online 对 JSON 数据进行了建模，并将其部署为模拟 API。

1.7 内容预告

第 2~4 章将介绍 JSON 在以下主流平台上的使用：

- JavaScript
- Ruby on Rails
- Java

在第 2 章中，你将利用之前用 `json-server` 所搭建的模拟 API 来学习如何在 JavaScript 中使用 JSON。

第2章

在JavaScript中使用JSON

我们在上一章中介绍了 JSON 数据交换格式的基础知识，本章则将开始用 JSON 来开发具体的应用程序。虽然 JSON 最初是 JavaScript 语言为对象和数组定义的一个子集，但如今它与 JavaScript 之间已经不存在绑定关系了。现在的 JSON 与具体编程语言无关，而且可以跨平台工作。不过，因为 JSON 源于 JavaScript，所以我们首先介绍 JSON 在 JavaScript 中的使用情况。

本章内容如下所示：

- 使用 `JSON.stringify()` 和 `JSON.parse()` 进行 JavaScript 中的序列化 / 反序列化操作；
- 使用 JavaScript 对象和 JSON；
- 调用 RESTful API，并使用 Mocha/Chai 对调用的结果进行单元测试；
- 搭建基于 JSON 的一个小型 Web 应用程序。

在本章的示例中，我们将会用到 Node.js，并用 Yeoman 来快速搭建 Web 应用程序，然后对上一章中用 `json-server` 所创建的 RESTful API 服务进行调用以获取数据。因为涉及的概念和内容很多，所以我们会逐个学习并应用。在开发 Web 应用程序前，先来了解一下 JavaScript 中的序列化 / 反序列化和对象方面的基础知识。

2.1 安装Node.js

正式开始前，可以参考 A.2 节中的操作说明来搭建开发环境。

2.2 用JSON.stringify()和JSON.parse()进行序列化/反序列化操作

为了以跨平台的方式向其他应用程序提供数据，一个应用程序需要将信息**序列化**为 JSON。同时，应用程序还必须能够**反序列化** JSON，从而将外部信息转换成自身可理解的数据结构。

2.2.1 用于stringify/parse操作的“JSON”对象

用于 `stringify/parse` 操作的“JSON”对象最初由 Douglas Crockford 所开发，并从 2009 年的 ECMAScript 5 标准开始成为 JavaScript 原生类库的一部分。该对象提供以下方法：

- `JSON.stringify()` 将信息序列化为 JSON；
- `JSON.parse()` 将 JSON 反序列化为 JavaScript 可以理解的数据结构。

另外，该对象还存在以下特征：

- 最初由 Douglas Crockford 所开发；
- 无法实例化；
- 除了 `stringify()` 和 `parse()`，不提供其他功能。

2.2.2 JavaScript中简单数据类型的JSON序列化操作

我们先来看一下对 JavaScript 中的下列基础数据类型的序列化操作：

- 数值
- 字符串
- 数组
- 布尔值
- 字面量对象

例 2-1 展示了如何使用 `JSON.stringify()` 来序列化简单数据类型。

例 2-1 js/basic-data-types-stringify.js

```
var age = 39; // 整型数
console.log('age = ' + JSON.stringify(age) + '\n');

var fullName = 'Larson Richard'; // 字符串
console.log('fullName = ' + JSON.stringify(fullName) + '\n');

var tags = ['json', 'rest', 'api', 'oauth']; // 数组
console.log('tags = ' + JSON.stringify(tags) + '\n');

var reqistered = true; // 布尔值
console.log('registered = ' + JSON.stringify(reqistered) + '\n');

var speaker = {
  firstName: 'Larson',
  lastName: 'Richard',
```

```
    email: 'larsonrichard@ecratic.com',
    about: 'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
    company: 'Ecratic',
    tags: ['json', 'rest', 'api', 'oauth'],
    registered: true
  };

  console.log('speaker = ' + JSON.stringify(speaker));
```

在命令行中用 node 运行上述文件后，可以得到以下结果：

```
json-at-work => node basic-data-types-stringify.js
age = 39

fullName = "Larson Richard"

tags = ["json","rest","api","oauth"]

registered = true

speaker = {"firstName":"Larson","lastName":"Richard","email":"larsonrichard@ecratic.com","about":"Incididunt mollit cupi
datat magna excepteur do tempor ex non ...","company":"Ecratic","tags":["json","rest","api","oauth"],"registered":true}
json-at-work =>
```

对于标量类型（数值、字符串、布尔值），JSON.stringify() 并没有提供什么有趣的功能。但对于 speaker 这样的对象字面量来说，JSON.stringify() 可以生成一段驳杂却合法的 JSON 字符串，因此显得比较有用。JSON.stringify() 还可以接受其他参数，从而让序列化操作变得更加强大。根据 Mozilla 开发者网络中的 JavaScript 指南，具体的语法如下所示。

```
JSON.stringify(value[, replacer [, space]])
```

参数列表如下所示。

value（必选）
: 需要进行序列化的 JavaScript 值。

replacer（可选）
: 函数或数组。如果是函数，则 stringify() 方法会为 value 对象中的每个名称 – 值对调用 replacer。

space（可选）
: 数值或字符串，表示缩进。如果是数值，则表示每一级缩进所占用的空格数。

例 2-2 展示了对 replacer 和 space 参数的使用，该示例优化了 speaker 对象的显示，同时也展示了对数据元素的过滤操作。

例 2-2　js/obj-literal-stringify-params.js

```
  var speaker = {
    firstName: 'Larson',
    lastName: 'Richard',
    email: 'larsonrichard@ecratic.com',
    about: 'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
    company: 'Ecratic',
    tags: ['json', 'rest', 'api', 'oauth'],
    registered: true
  };
```

```
function serializeSpeaker(key, value) {
  return (typeof value === 'string' || Array.isArray(value)) ? undefined : value;
}

// 优化显示结果。
console.log('Speaker (pretty print):\n' + JSON.stringify(speaker, null, 2) + '\n');


// 过滤掉数据中的字符串和数组，优化显示结果。
console.log('Speaker without Strings and Arrays:\n' +
  JSON.stringify(speaker, serializeSpeaker, 2));
```

运行以上文件可以得到以下结果：

```
json-at-work => node obj-literal-stringify-params.js
Speaker (pretty print):
{
  "firstName": "Larson",
  "lastName": "Richard",
  "email": "larsonrichard@ecratic.com",
  "about": "Incididunt mollit cupidatat magna excepteur do tempor ex non ...",
  "company": "Ecratic",
  "tags": [
    "json",
    "rest",
    "api",
    "oauth"
  ],
  "registered": true
}

Speaker without Strings and Arrays:
{
  "registered": true
}
json-at-work =>
```

在以上示例中，第一次 `JSON.stringify()` 调用将缩进设置为 2 个空格，从而优化了 JSON 的输出结果。第二次调用则使用 `serializeSpeaker()` 函数作为 `replacer` 参数（JavaScript 中的函数可以作为表达式以参数的形式进行传递）。`serializeSpeaker()` 会检查每个值的类型，如果类型为字符串或数组，则返回 `undefined`，其余情况则返回值本身。

对于 `undefined` 值，`JSON.stringify()` 的处理方式如下所示：

- 如果 `undefined` 是对象中的某个字段的值，则序列化时直接忽略该字段；
- 如果 `undefined` 是数组中的一个值，则将其转换为 `null`。

2.2.3 使用toJSON()进行对象的序列化操作

正如你在前面看到的，JSON 序列化操作最适合应用于对象。可以通过向 `speaker` 对象中添加 `toJSON()` 方法来自定义 `JSON.stringify()` 的输出结果，如例 2-3 所示。

例 2-3　js/obj-literal-stringify-tojson.js

```
var speaker = {
  firstName: 'Larson',
  lastName: 'Richard',
  email: 'larsonrichard@ecratic.com',
  about: 'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
  company: 'Ecratic',
  tags: ['json', 'rest', 'api', 'oauth'],
  registered: true
};

speaker.toJSON = function() {
  return "Hi there!";
}

console.log('speaker.toJSON(): ' + JSON.stringify(speaker, null, 2));
```

序列化操作的执行结果如下所示：

```
json-at-work => node obj-literal-stringify-tojson.js
speaker.toJSON(): "Hi there!"
json-at-work => 
```

如果一个对象拥有 `toJSON()` 方法，则 `JSON.stringify()` 不再试图将其转换成字符串，而是直接输出该对象的 `toJSON()` 方法所返回的值。使用 `toJSON()` 方法是合法的，但不一定明智。因为一旦使用了 `toJSON()` 方法，开发者就必须自行承担对整个对象结构的序列化操作，而这完全违背了 `JSON.stringify()` 方法的初衷。对于 `speaker` 这样简单的对象来说，可能没有什么问题，但对于包含其他对象的复杂对象来说，相关的序列化代码迟早会变得非常庞杂。

2.2.4　使用eval()进行JSON的反序列化操作

JavaScript 开发者最初是使用 `eval()` 函数来解析 JSON 的。`eval()` 可以接受一个字符串作为参数，该字符串可以是一个 JavaScript 表达式、一句 JavaScript 语句，或者一连串的 JavaScript 语句，参见例 2-4。

例 2-4　js/eval-parse.js

```
var x = '{ "sessionDate": "2014-10-06T13:30:00.000Z" }';

console.log('Parse with eval(): ' + eval('(' + x + ')').sessionDate + '\n');

console.log('Parse with JSON.parse(): ' + JSON.parse(x).sessionDate);
```

运行以上文件可以得到以下结果：

```
json-at-work => node eval-parse.js
Parse with eval(): 2014-10-06T13:30:00.000Z

Parse with JSON.parse(): 2014-10-06T13:30:00.000Z
json-at-work => 
```

在这一示例中，`eval()` 和 `JSON.parse()` 的执行结果相同，都解析了日期属性。那么 `eval()` 的问题在哪里？例 2-5 展示了在字符串中包含 JavaScript 语句的一个示例。

例 2-5　js/eval-parse-2.js

```
var x = '{ "sessionDate": new Date() }';

console.log('Parse with eval(): ' + eval('(' + x + ')').sessionDate + '\n');

console.log('Parse with JSON.parse(): ' + JSON.parse(x).sessionDate);
```

运行以上代码可以得到以下结果：

```
tmarrs => node eval-parse-2.js
Mon Oct 06 2014 20:54:18 GMT-0600 (MDT)

undefined:1
{ "sessionDate": new Date() }
                 ^
SyntaxError: Unexpected token e
    at Object.parse (native)
    at Object.<anonymous> (/Users/tmarrs/projects/json-at-work/chapter-2/js/eval-parse-2.js:5:18)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:906:3
tmarrs =>
```

上述示例的文本参数中含有 `new Date()` 这一 JavaScript 语句，而 `eval()` 成功执行了这一语句。与此同时，`JSON.parse()` 则正确地拒绝了这一文本参数，并报错显示传入的 JSON 是非法的。在这一示例中，嵌入的语句只是创建了一个 `Date` 对象，并无多少危害；但其他嵌入语句可能包含恶意代码，而 `eval()` 会照样执行。因此，尽管 `eval()` **可以**用于解析 JSON，但该做法一般并不安全，也不合适。`eval()` 会打开允许任何 JavaScript 表达式进入的后门，这会让应用程序更易受到攻击。由于这一安全问题，对 JSON 的解析已经废弃了 `eval()` 函数，取而代之的是 `JSON.parse()`。

2.2.5　使用`JSON.parse()`进行JSON的反序列化操作

我们回到演讲者数据这一示例，例 2-6 展示了使用 `JSON.parse()` 将 JSON 字符串反序列化为 `speaker` 对象的情况。

例 2-6　js/obj-literal-parse.js

```
var json = '{' + // 以多行形式定义的JSON字符串。
  '"firstName": "Larson",' +
  '"lastName": "Richard",' +
  '"email": "larsonrichard@ecratic.com",' +
  '"about": "Incididunt mollit cupidatat magna excepteur do tempor ex non ...",' +
  '"company": "Ecratic",' +
  '"tags": [' +
    '"json",' +
    '"rest",' +
    '"api",' +
    '"oauth"' +
  '],' +
  '"registered": true' +
'}';
```

```
// 将JSON字符串反序列化为speaker对象。
var speaker = JSON.parse(json);

// 打印speaker对象。
console.log('speaker.firstName = ' + speaker.firstName);
```

运行以上文件可以得到以下结果：

```
json-at-work => node obj-literal-parse.js
speaker.firstName = Larson
json-at-work =>
```

`JSON.parse()` 接受一个 JSON 字符串作为输入，并将其解析为完整的 JavaScript 对象。在以上示例中，解析后即可访问 `speaker` 对象的数据字段。

2.3 JavaScript对象和JSON

到目前为止，我们展示了 JavaScript 中的核心数据类型和简单的字面量对象是如何与 JSON 进行交互的。对于这一过程中所忽略的诸多细节，接下来将一一深入阐述。在创建（或**实例化**）JavaScript 对象的几种方式中，对象字面量形式与 JSON 对象最为贴近，因此之后的内容也将主要采用字面量形式来创建对象。

例 2-7 再一次展示了以对象字面量形式表示的 `speaker` 对象。

例 2-7 js/obj-literal.js

```
var speaker = {
  firstName: 'Larson',
  lastName: 'Richard',
  email: 'larsonrichard@ecratic.com',
  about: 'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
  company: 'Ecratic',
  tags: ['json', 'rest', 'api', 'oauth'],
  registered: true,
  name: function() {
    return (this.firstName + ' ' + this.lastName);
  }
};
```

在对象字面量语法中，对象的属性（数据和函数）会在大括号中直接定义。在以上示例中，`speaker` 对象被赋值并实例化。因为对象字面量在组合对象的数据和功能上简洁而又不失模块化，所以如果在应用程序中无须再次创建另一个 `speaker` 对象实例时，则使用对象字面量创建对象会是一个很不错的方案。对象字面量的真正缺陷在于每次使用字面量时只能创建一个 `speaker` 实例，同时实例中的 `name()` 方法也无法重用。

2.3.1 Node REPL

到目前为止，我们都是在命令行中以执行 JavaScript 文件的方式来使用 Node.js。接下来我们转换方式，看看 Node.js 的解释器，即读取—求值—输出—循环（Request-Eval-Print-Loop，REPL）。REPL 是一个非常不错的工具，它提供了对代码的即时执行与反馈，方便反复调试代码并改善应用程序。在 Node.js 的官方文档中可以找到有关 REPL 的深入描述。

与其他工具一样，REPL 也不是完美的。其中一处地方尤其不便：

```
json-at-work => node
> var x = 0;
undefined
> var y = x + 5;
undefined
> .exit
json-at-work =>
```

对于不产生输出的每个语句，解释器执行后将输出 `undefined`。很多人认为这一点容易分散注意力，而事实上也确实有方法来禁止此行为。可以参见 A.2 节中的“改造 REPL——`mynode`”部分的内容来配置 `mynode` 这一命令行别名，我认为 `mynode` 比标准的 Node.js REPL 更易用。

言归正传，下面来看看 `speaker` 对象在 `mynode` REPL 中的使用情况：

```
json-at-work => mynode
> var speaker = {
...    firstName: 'Larson',
...    lastName: 'Richard',
...    email: 'larsonrichard@ecratic.com',
...    about: 'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
...    company: 'Ecratic',
...    tags: ['json', 'rest', 'api', 'oauth'],
...    registered: true,
...    name: function() {
.....      return (this.firstName + ' ' + this.lastName);
.....    }
... };
>
> speaker
{ firstName: 'Larson',
  lastName: 'Richard',
  email: 'larsonrichard@ecratic.com',
  about: 'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
  company: 'Ecratic',
  tags:
   [ 'json',
     'rest',
     'api',
     'oauth' ],
  registered: true,
  name: [Function] }
>
> speaker.name();
'Larson Richard'
> .exit
json-at-work =>
```

从以上的运行结果中可以看到，我们能够直接与 `speaker` 对象交互，调用其方法并在解释器中查看调用结果。

以下是 REPL 中可以使用的一些命令。

`.clear`

清除当前 REPL 中的上下文。

`.break`

返回 REPL 提示符。可以在多行语句中使用此命令来快速退回提示符。

`.exit`

退出当前 REPL。

`.save`

将当前 REPL 保存为文件。

2.3.2 有关JavaScript对象的更多学习资料

之前的内容忽略了 JavaScript 面向对象编程方面的很多细节，而事实上在 JavaScript 中还存在着其他一些与对象进行交互的方法。在面向对象方面，之前的介绍只涉及了一些皮毛，这些内容仅能够让我们在应用程序中以合理的方式使用 JavaScript 对象与 JSON。完整、深入地介绍 JavaScript 对象超出了本书的探讨范围。如果需要深入理解这方面的知识，可以参考以下高质量资源：

- JD Isaacks 所著的 *Learn JavaScript Next*（Manning 出版社）；
- Nicholas C. Zakas 所著的《JavaScript 面向对象精要》[1]；
- Addy Osmani 所著的《JavaScript 设计模式》[2]。

2.4 用模拟API进行单元测试

了解了如何对 `speaker` 对象进行序列化 / 反序列化操作后，我们可以运行一个简单的服务器端单元测试程序，以便对 `json-server` 提供的模拟 API 进行测试。而在之后所创建的小 Web 应用程序中，该模拟 API 也会得到使用。

2.4.1 单元测试风格——TDD和BDD

测试驱动开发（Test-Driven Development，TDD）是使用单元测试来驱动开发工作的一种开发策略。其典型工作流程如下。

(1) 编写一些测试用例。
(2) 运行测试用例，因为还没有任何实际代码，所以运行结果会失败。
(3) 编写实际代码，可以通过测试就好。
(4) 重构实际代码，改进代码的设计与灵活性。
(5) 重新运行测试，修复出现的问题，直至所有测试用例通过。

TDD 风格的单元测试呈现出过程式的特点。

行为驱动开发（Behavior-Driven Development，BDD）是一种根据验收标准和期望结果来测试用户故事的开发策略。BDD 风格的测试用例读起来和普通的句子相同，例如，“演讲者应当在 30 天之内从会议方收到他们的报酬”。如需了解更多有关 BDD 的信息，可以参考 Dan North 的一篇非常不错的文章“Introducing BDD”。有些人认为 BDD 只是 TDD 的一种改进。我倾向于同意这一观点，因为开发者在 BDD 过程中会遵循和 TDD 一样的工作流程。

注 1：此书已由人民邮电出版社出版。——编者注
注 2：此书已由人民邮电出版社出版。——编者注

BDD 和 TDD 都是可靠的开发策略，并且可以结合起来使用，从而为应用程序构造健壮的测试套件。本章的单元测试中的断言将使用 BDD 风格编写。

2.4.2 使用Mocha和Chai即可完成单元测试

接下来的服务器端单元测试将会用到以下两个工具。

Mocha
: Mocha 是一个 JavaScript 单元测试框架，在 Node.js 和浏览器中均可运行。本节会在 Node.js 项目中用命令行的方式使用 Mocha，并添加功能来实现基于 JSON 的 API 测试。如需了解更多有关 Mocha 的细节，可访问其官方网站。

Chai
: Chai 是一个断言库，用于在 JavaScript 测试框架（如本节中的 Mocha）中添加更丰富的断言语句，从而对框架形成有益的补充。Chai 允许开发者以 TDD 或者 BDD 的风格来编写测试。本章的测试用例将使用 `expect`（BDD）的断言风格，但你也可以尝试一下 `should`（BDD）或者 `assert`（TDD）的断言风格。总之，采用自己觉得顺手的风格即可。如需了解更多有关 Chai 的细节，可访问其官方网站。

2.4.3 设置单元测试环境

在进一步深入前，需要确保已设置好测试环境。如尚未安装 Node.js，可参考 A.2 节和 A.2.5 节的内容来执行 Node.js 的安装操作。如需依照本节的描述运行代码示例中的项目，可使用 `cd` 命令切换到 chapter-2/speakers-test 目录，并执行以下命令来安装项目依赖：

```
npm install
```

如需手动创建本节中的 Node.js 项目，可参考本书在 GitHub 上的相关指导步骤。

2.4.4 Unirest

本节的单元测试需要通过 HTTP 来调用 API，因此我们会在编写测试的过程中使用 Unirest。Unirest 是 Mashape 团队开发的一款开源跨平台的 REST 客户端工具，涵盖 JavaScript、Node.js、Ruby on Rails 和 Java 等多种编程语言。Unirest 简单易用，在对 REST API 进行 HTTP 调用的所有客户端代码中都能很好地工作，与此同时，它还非常适用于单元测试。对于测试套件中的 HTTP 配置信息（URI、HTTP 头部等），使用 Unirest 可以做到一次配置、多次调用，因此有利于编写更加整洁的单元测试用例。如需了解 Unirest 的详细文档，可访问其官方网站。

跨平台性与概念和调用方法在多种编程语言实现间的高度相似，使得 Unirest 成为了非常不错的选择。除此之外，也存在其他一些优秀的 HTTP 类库，有 Web 的、移动端的，以及基于 Java 的（如 Apache 的 HttpClient，不过作为多门编程语言的开发者，我更倾向于使用 Unirest。值得一提的是，Unirest 不仅适用于单元测试，作为 HTTP 客户端领域的 API 封装程序，Unirest 有着广泛的应用）。

2.4.5　测试数据

本节将使用第 1 章中提供的演讲者数据作为测试数据，并将其部署为 RESTful API。与第 1 章相同，我们将使用 `json-server` 这个 Node.js 模块把 data/speakers.json 文件部署为 Web API。如需了解有关安装 `json-server` 的信息，可参考 A.2.5 节中的内容。

以下是在本地机器中使用 5000 端口运行 `json-server` 的步骤：

```
cd chapter-2/data

json-server -p 5000 ./speakers.json
```

2.4.6　对演讲者数据进行单元测试

例 2-8 中的单元测试展示了如何使用 Unirest 向 `json-server` 提供的模拟 API 发起调用。

例 2-8　speakers-test/speakers-spec.js

```
'use strict';

var expect = require('chai').expect;
var unirest = require('unirest');

var SPEAKERS_ALL_URI = 'http://localhost:5000/speakers';


describe('speakers', function() {
  var req;

  beforeEach(function() {
    req = unirest.get(SPEAKERS_ALL_URI)
      .header('Accept', 'application/json');
  });

  it('should return a 200 response', function(done) {
    req.end(function(res) {
      expect(res.statusCode).to.eql(200);
      expect(res.headers['content-type']).to.eql(
        'application/json; charset=utf-8');

      done();
    });
  });

  it('should return all speakers', function(done) {
    req.end(function(res) {
      var speakers = res.body;
      var speaker3 = speakers[2];

      expect(speakers.length).to.eql(3);
      expect(speaker3.company).to.eql('Talkola');
      expect(speaker3.firstName).to.eql('Christensen');
      expect(speaker3.lastName).to.eql('Fisher');
      expect(speaker3.tags).to.eql([
        'Java', 'Spring',
        'Maven', 'REST'
```

```
      ]);

      done();
    });
  });

});
```

该单元测试中进行了以下操作。

- 测试代码在 Mocha 的 `beforeEach()` 方法中对 `unirest` 的 URI 和 `Accept` 头部进行配置，从而避免了配置代码的重复。在 `describe` 语句所定义的范围内，Mocha 会在执行每个测试用例（即 `it` 语句）前先运行一次 `beforeEach()` 方法。
- 测试代码中的 `should return all speakers` 测试用例最有意思，其工作机制如下。
  - `req.end()` 异步地执行 Unirest 中的 `GET` 请求，并使用匿名函数来处理 API 调用所返回的 HTTP 响应（`res`）。
  - 测试用例从 HTTP 响应体（`res.body`）中提取 `speakers` 对象。此时，Unirest 已经自动解析了 API 所返回的 JSON，并以对象字面量的形式转换成对应的 JavaScript 对象。
  - 测试用例使用 Chai 中 BDD 风格的 `expect` 断言语句来检查预期结果。
    - ♦ 存在 3 个 `speaker`。
    - ♦ 第三个 `speaker` 的 `company`、`firstName`、`lastName` 和 `tags` 字段与 speakers.json 文件中的值保持一致。

可以新开命令行终端并执行以下命令来运行上面的单元测试程序：

```
cd chapter-2/speakers-test

npm test
```

运行后可观察到以下结果：

```
json-at-work => npm test

...

> mocha test

...

  speakers
    ✓should return a 200 response
    ✓should return all speakers


  2 passing
```

2.5　搭建小型Web应用程序

在介绍了如何对 `speaker` 对象进行 JSON 序列化 / 反序列化操作，以及如何对 `json-server` 所提供的模拟 API 进行单元测试后，现在是时候搭建一个简单的 Web 应用将 API 数据呈现给用户了。

这一 Web 应用的开发过程分为 3 个阶段。

- 第 1 阶段：使用 Yeoman 生成一个基本的 Web 应用。
- 第 2 阶段：使用 jQuery 发起 HTTP 请求。
- 第 3 阶段：在模板中使用 `json-server` 提供的模拟 API 数据。

2.5.1 Yeoman

Yeoman 与 Java 社区中的 Gradle、Maven 以及 Ruby on Rails 比较像，它提供了一种简单的方式来快速创建 Web 应用程序，从而简化开发者的工作流程。本节将使用 Yeoman 来搭建、开发并运行示例应用程序。如需安装（依赖 Node.js 的）Yeoman，可参考 A.2.4 节中的指导步骤。

Yeoman 可以提供以下功能：

- 创建开发环境；
- 运行应用程序；
- 保存代码改动后自动刷新浏览器；
- 管理包依赖关系；
- 压缩应用程序的代码并打包，以便于部署。

Yeoman 遵循"约定优于配置"的理念：

- 自动化搭建；
- 可以工作就好；
- 使用标准目录结构；
- 提供依赖管理；
- 预设合理的默认值；
- 鼓励最佳实践；
- 基于工具的开发流程（如在测试、代码检查、运行和打包过程中使用专业的工具）。

如需了解更多有关 Yeoman 的信息，可参考以下 Yeoman 教程：

- 使用 Yeoman 快速搭建 Web 应用程序；
- 使用 Yeoman 工作流构建应用程序。

1. Yeoman工具集

Yeoman 由以下工具组成。

快速搭建

  可以使用 Yo 命令来生成应用程序的目录结构及 Grunt/Gulp/Bower 的配置文件。

构建

  可以使用 Gulp 或 Grunt 来构建、运行、测试和打包应用程序。

包管理

  可以使用 Bower 或 npm 来管理和下载包依赖。

虽然 Grunt 是一个可靠的构建工具，npm 也是一个非常优秀的包管理器，但因为 Yeoman 的 Web 应用程序生成器中使用了 Gulp 和 Bower，所以我们也将在本节示例中使用 Gulp 和 Bower。

2. Yeoman生成器

Yeoman 使用生成器来快速构建项目。每个生成器都可以创建一个预先配置好的模板应用程序。对于目前已有的上千个生成器，Yeoman 官方提供了一份完整的列表。

2.5.2　第1阶段：使用Yeoman生成Web应用程序

首先，我们创建一个不包含任何实际功能的简单应用程序，并将演讲者数据硬编码到一个表格中。我们会在第 2 和第 3 阶段中添加真实的演讲者数据。安装好 Yeoman 后，我们将使用 `generator-webapp` 生成器来创建开箱即用的 Web 应用程序，该应用程序自带网页、CSS 样式、Bootstrap 4、jQuery、Mocha 和 Chai。

如需手动创建本节中的 Yeoman 项目，可参考本书在 GitHub 上的相关指导步骤。如需依照本节的描述运行代码示例中的 Yeoman 项目，可使用 `cd` 命令切换到 chapter-2/speakers-web-1 目录。无论何种形式，都可以使用以下命令来启动应用程序：

```
gulp serve
```

这一命令可以启动本地 Web 服务器，并在默认浏览器中打开首页（index.html）。可以通过 http://localhost:9000 访问到以下页面，如图 2-1 所示。

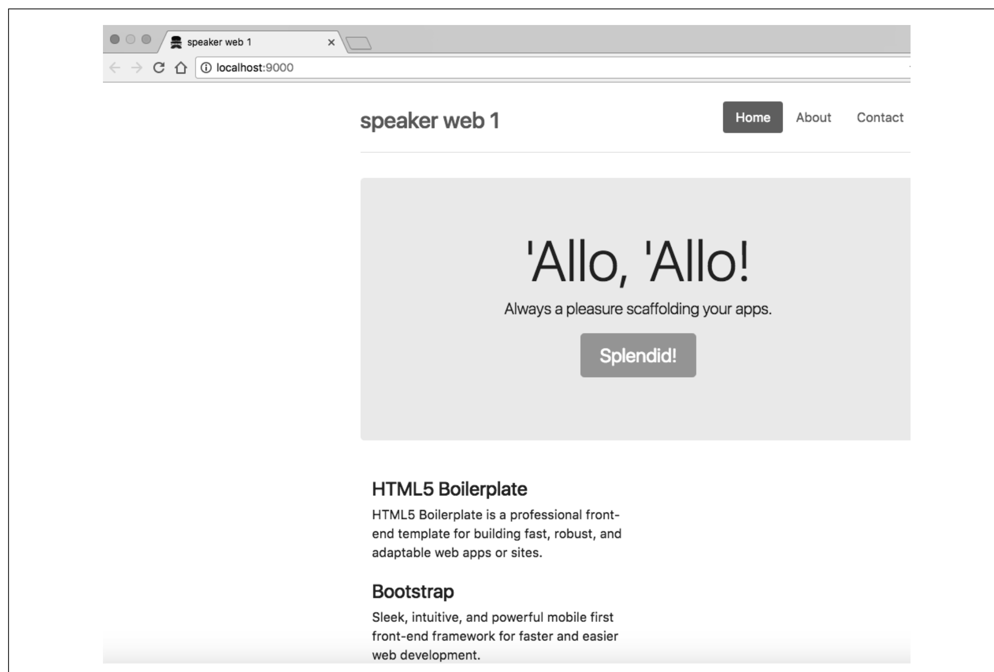


图 2-1：用 Yeoman 生成的基础 Web 应用程序

值得注意的是，如果应用程序保持运行，那么保存代码修改后，应用程序会使用 LiveReload 来自动更新页面上的相关内容。

使用 Yeoman 的 `generator-webapp` 生成器创建好初始应用程序后，就可以对该应用程序进行定制了。首先，可以修改 index.html 页面中的标题、页眉和显示屏组件（即移除“Splendid!”按钮），如例 2-9 所示。

例 2-9　speakers-web-1/app/index.html

```
<!doctype html>
<html lang="">
  <head>

   ...

    <title>JSON at Work - MyConference</title>

    ...

  </head>
  <body>
    ...

    <div class="header">
      ...

      <h3 class="text-muted">JSON at Work - Speakers</h3>
    </div>

    ...

    <div class="jumbotron">
      <h1 class="display-3">Speakers</h1>
      <p class="lead">Your conference lineup.</p>
    </div>

    ...

  </body>
</html>
```

然后在 index.html 文件中以硬编码的方式添加演讲者数据表格，如例 2-10 所示。

例 2-10　speakers-web-1/app/index.html

```
<!doctype html>
<html lang="">

  ...

  <body>

    ...

    <table class="table table-striped">
      <thead>
        <tr>
          <th>Name</th>
          <th>About</th>
          <th>Topics</th>
```

```
          </tr>
        </thead>
        <tbody id="speakers-tbody">
          <tr>
            <td>Larson Richard</td>
            <td>Incididunt mollit cupidatat magna excepteur do tempor ...
            </td>
            <td>JavaScript, AngularJS, Yeoman</td>
          </tr>
          <tr>
            <td>Ester Clements</td>
            <td>Labore tempor irure adipisicing consectetur velit. ...
            </td>
            <td>REST, Ruby on Rails, APIs</td>
          </tr>
          <tr>
            <td>Christensen Fisher</td>
            <td>Proident ex Lorem et Lorem ad. Do voluptate officia ...
            </td>
            <td>Java, Spring, Maven, REST</td>
          </tr>
        </tbody>
      </table>

      ...

    </body>
  </html>
```

这样就可以实现一个显示示例演讲者数据的 Web 应用程序，如图 2-2 所示。

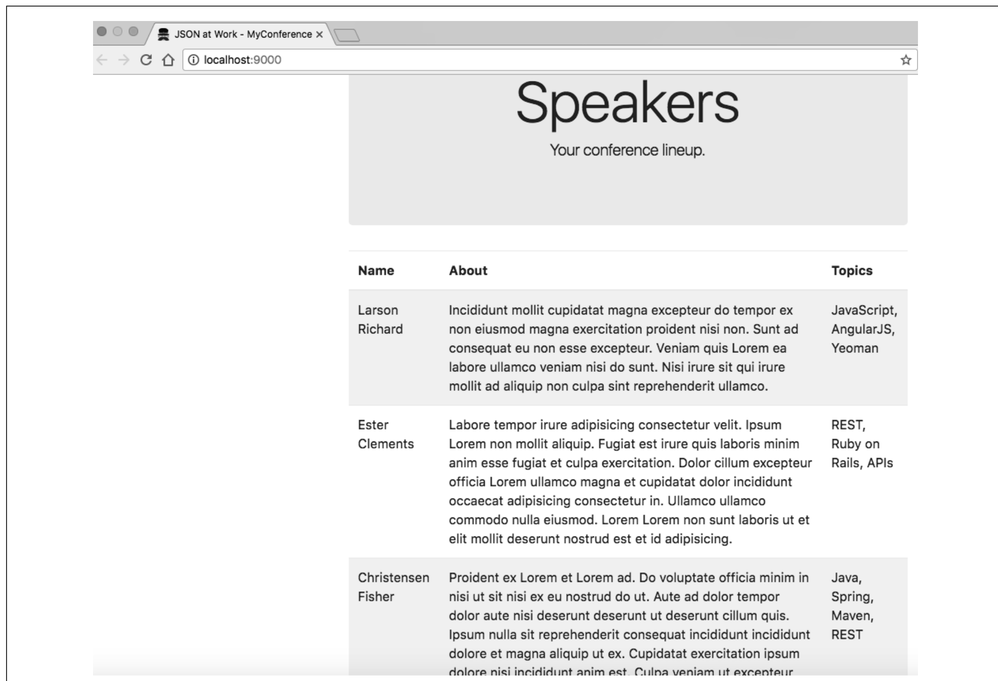


图 2-2：index.html 中的示例演讲者数据

以下是 `generator-webapp` 生成的应用程序中比较关键的文件和目录。

- app/ 目录中包含应用程序的代码（如 HTML、JavaScript 和 CSS）。
  - index.html 为应用程序的主页。
  - images/ 目录包含应用程序所使用的图片文件。
  - scripts/ 目录包含应用程序所使用的 JavaScript 等脚本文件。
    - main.js 是应用程序的主 JavaScript 文件。我们将在第 2 阶段中详细介绍此文件。
  - styles/ 目录包含 CSS 等样式文件。
- bower_components/ 目录包含由 Bower 所安装的项目依赖：Bootstrap、jQuery、Mocha 和 Chai。
- node_modules/ 目录包含 Gulp 等 Node.js 所需要的项目依赖。
- test/ 目录包含测试框架所使用的测试用例。本例使用的测试工具为 Mocha 和 Chai。
- gulpfile.js 是 Gulp 的构建脚本文件，用于构建和运行应用程序。
- package.json 是 Node.js 用于管理依赖的配置文件，Gulp 则会使用这些依赖来运行项目脚本。
- dist/ 目录包含由 `gulp build` 命令所构建的文件。

关于使用 `generator-webapp` 生成的应用程序，以下是一些有用的重要命令。

Ctrl-C
  关闭 Web 服务器，退出应用程序。

`gulp lint`
  使用 `lint` 对应用程序中的 JavaScript 文件进行语法检查。

`gulp +serve:test`
  对 Web 应用程序进行测试。在本例中，测试程序会在 PhantomJS 环境中使用 Mocha 和 Chai。

`gulp build`
  对应用程序进行构建、打包以方便部署。

`gulp clean`
  清除测试和构建应用程序过程中所生成的文件。

可以在命令行中输入 `gulp --tasks` 来获取完整的命令列表。

开始第 2 阶段的工作前，确保关闭之前所启动的 Web 应用程序。

2.5.3　第2阶段：使用jQuery发起HTTP请求

第 1 阶段中开发的 Web 应用程序以硬编码的方式显示演讲者数据，而现在是时候添加“真实的”内容和功能了。

具体工作分为以下 3 个步骤。

(1) 重构首页代码，移除硬编码的演讲者数据。
(2) 添加单独的 JSON 文件来保存演讲者数据。

(3) 使用 jQuery 将 JSON 文件中的演讲者数据显示在页面上。

如需自行搭建第 2 阶段中的 Yeoman 项目，可以执行以下操作。

- 参考本书在 GitHub 上的相关指导步骤。
- 记得将第 1 阶段中修改后的 app/index.html 文件复制到项目中。

如需依照本节的描述运行代码示例中的 Yeoman 项目，可以使用 `cd` 命令切换到 chapter-2/speakers-web-2 目录。无论何种形式，都可以使用以下命令来启动应用程序：

```
gulp serve
```

和第 1 阶段一样，这一命令可以启动本地 Web 服务器。可通过 http://localhost:9000 访问主页，如图 2-3 所示。

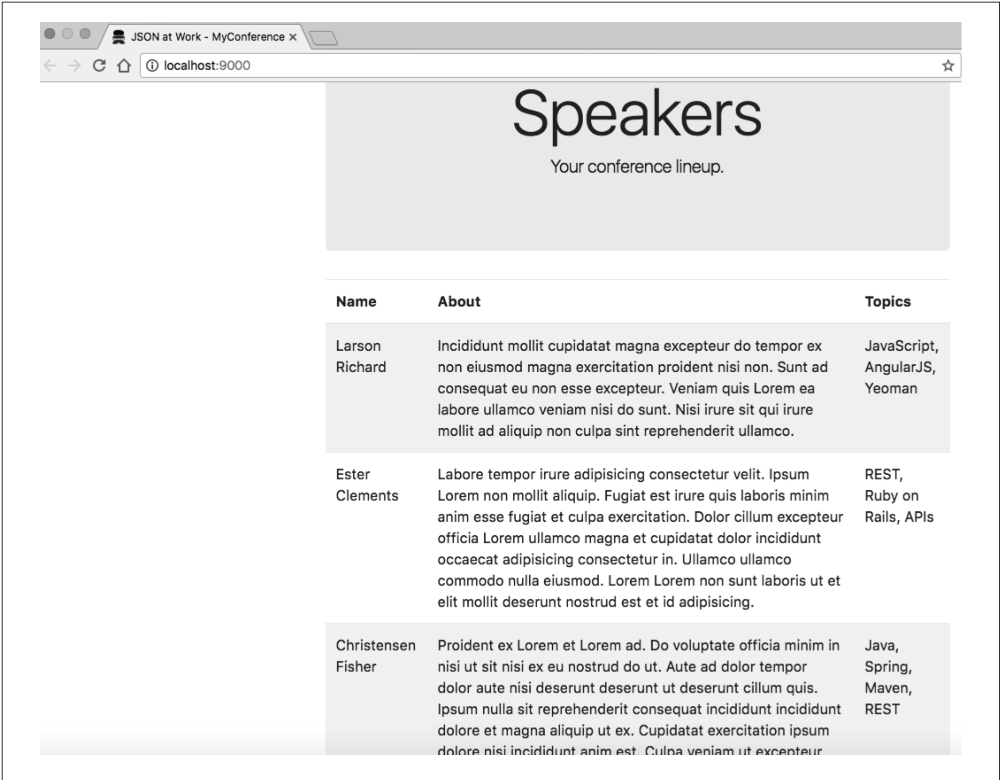


图 2-3：示例演讲者数据

和之前看到的结果一样，主页（index.html）在表格中展示了硬编码的演讲者数据。请保持应用程序运行，以观察代码修改的结果。

接下来，我们移除表格内容中所有行的数据。修改后演讲者表格的 HTML 代码如例 2-11 所示。

例 2-11 speakers-web-2/app/index.html

```
<!doctype html>
<html lang="">

  ...

  <body>

    ...

    <table class="table table-striped">
      <thead>
        <tr>
          <th>Name</th>
          <th>About</th>
          <th>Topics</th>
        </tr>
      </thead>
      <tbody id="speakers-tbody">
      </tbody>
    </table>

    ...

  </body>
</html>
```

例 2-11 中包含了一个只有标题行的空表格。该表格使用 Bootstrap 的 `table-striped` CSS 类让内容行呈现出灰白相间的效果。注意，用于存储表格内容的 `<tbody>` 元素定义了名为 `speakers-tbody` 的 ID，而 jQuery 就是通过这个 ID 来动态添加表格行的。

接下来，我们需要在一个单独的 JSON 文件中存储演讲者数据。参见从 /chapter-2/data/speakers.json 处复制而来的带有演讲者数据的 /speakers-web-2/app/data/speakers.json 文件。

最后，修改 app/scripts/main.js 文件，使用 jQuery 将 app/data/speakers.json 文件中的数据添加到演讲者表格中，如例 2-12 所示。

例 2-12 speakers-web-2/app/scripts/main.js

```
'use strict';

console.log('Hello JSON at Work!');

$(document).ready(function() {

  function addSpeakersjQuery(speakers) {
    $.each(speakers, function(index, speaker) {
```

```
        var tbody = $('#speakers-tbody');
        var tr = $('<tr></tr>');
        var nameCol = $('<td></td>');
        var aboutCol = $('<td></td>');
        var topicsCol = $('<td></td>');
        nameCol.text(speaker.firstName + ' ' + speaker.lastName);
        aboutCol.text(speaker.about);
        topicsCol.text(speaker.tags.join(', '));

        tr.append(nameCol);
        tr.append(aboutCol);
        tr.append(topicsCol);
        tbody.append(tr);
      });
    }

    $.getJSON('data/speakers.json',
      function(data) {
        addSpeakersjQuery(data.speakers);
      }
    );

  });
```

在以上示例中，前端代码包含在 jQuery 的 `$(document).ready()` 里，因此会等包括 DOM 在内的整个页面完全加载后才运行。代码中的 `$.getJSON()` 是一个 jQuery 方法，用于向某个 URL 发送 HTTP `GET` 请求，并将 JSON 响应转换为 JavaScript 对象。因为示例中的 app/data/speakers.json 文件被部署为 Web 应用程序的一部分，所以可以通过 HTTP 协议以 URL 的形式访问。最终，获得演讲者数据后，`$.getJSON()` 的回调函数会将填充演讲者表格的工作交给 `addSpeakersjQuery()` 函数来进行。

`addSpeakersjQuery()` 函数使用 jQuery 提供的 `.each()` 方法来遍历包含演讲者数据的数组。`.each()` 函数进行的工作包括以下几项：

- 使用 index.html 文件中的 `speakers-tbody` ID 来找到演讲者表格中的 `<tbody>` 元素；
- 使用 `speaker` 对象中的数据来填充 `<tr>` 和 `<td>` 元素，从而创建相应的表格行及其列；
- 将新创建的表格行添加到 `<tbody>` 元素中。

如需了解更多有关 jQuery 的 `getJSON()` 函数的信息，可参考 jQuery 官方文档。

如果之前一直保持应用程序运行，那么现在可以看到如图 2-4 所示的截图。

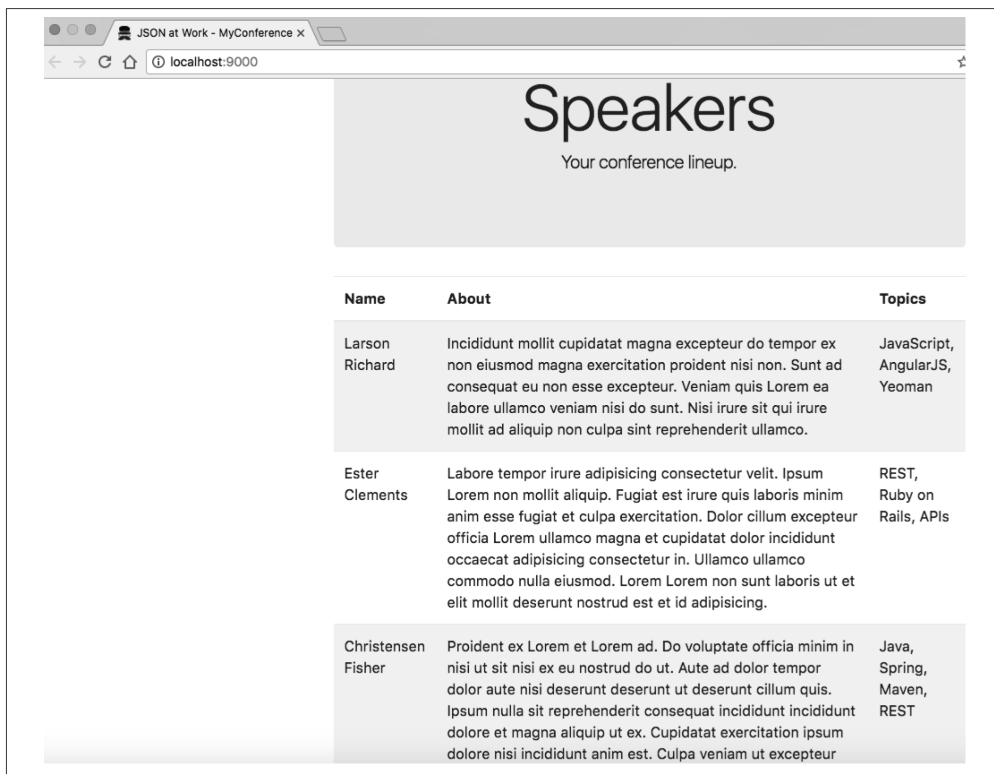


图 2-4：用 jQuery 和 JSON 文件所得到的示例演讲者数据结果

正如所预期的那样，页面看上去和本节刚开始时一模一样。但与一开始不同的是，重构后移除了页面中硬编码的演讲者数据，并以 HTTP 请求取而代之。到目前为止，我们已经有了一个真正的 Web 应用程序的雏形：动态填充页面。但这一雏形依旧存在以下不足。

- JSON 数据由 Web 应用程序中的静态文件所提供，我们则希望从 RESTful API 中来获取数据。
- JavaScript 代码与页面中的 HTML 元素耦合过紧，我们则希望减少 HTML 和 DOM 的操作。

开始第 3 阶段的工作前，确保关闭了之前所启动的 Web 应用程序。

2.5.4　第3阶段：在模板中使用模拟API所提供的演讲者数据

第 2 阶段中开发的 Web 应用程序通过发送 HTTP 请求将 JSON 文件中的演讲者数据填充到主页上，接下来的内容则会从 `json-server` 所提供的模拟 API 中来获取数据。除此之外，我们还会重构 JavaScript 代码，移除有关 HTML 和 DOM 的操作，并以 Mustache 这一外部模板取而代之。

具体工作分为以下 2 步。

(1) 修改 HTTP 请求，将请求发送至由 `json-server` 所提供的 URI 中。
(2) 使用 Mustache 模板，移除 JavaScript 代码中的 HTML 和 DOM 操作。

如需自行搭建第 3 阶段中的 Yeoman 项目，可以执行以下操作。

- 参考本书在 GitHub 上的相关指导步骤。
- 记得将第 2 阶段中修改的以下文件复制到项目中。
  - app/index.html
  - app/scripts/main.js

如需依照本节的描述运行代码示例中的 Yeoman 项目，可以使用 `cd` 命令切换到 chapter-2/speakers-web-3 目录。

接下来，我们修改 main.js 中的 HTTP 请求，将请求发送至由 `json-server` 所提供的模拟演讲者数据 API 中，如例 2-13 所示。

例 2-13 speakers-web-3/app/scripts/main.js

```
...

  $.getJSON('http://localhost:5000/speakers',
    function(data) {
      addSpeakersjQuery(data);
    }
  );

...
```

修改后，代码将调用由 `json-server` 所提供的模拟演讲者数据 API。值得注意的是，因为 `json-server` 所提供的数据不包含 `speakers` 字段，所以传入 `addSpeakersjQuery()` 的参数是 `data` 而不是第 2 阶段中的 `data.speakers`。

运行 Web 应用程序前，打开一个新的命令行窗口，在 5000 端口上运行 `json-server`：

```
cd chapter-2/data

json-server -p 5000 ./speakers.json
```

然后在另一个新的命令行窗口中运行以下命令，以启动 Web 应用程序：

```
gulp serve
```

与第 1、第 2 阶段一样，这一命令会启动本地 Web 服务器。可以在浏览器中通过访问 http://localhost:9000 看到与之前一样的演讲者数据。虽然结果相同，但因为使用的数据来自于 API 而不是静态文件，所以 Web 应用程序的质量比之前要好得多。请保持应用程序运行，以观察代码修改的结果。

接下来我们重构 JavaScript 代码，移除其中的 HTML 和 DOM 操作并使用 Mustache 模板。Mustache 号称自己提供的是**无逻辑模板**，即在使用 JavaScript 等语言来生成 HTML 的过程中无须使用控制语句（`for`、`if` 等）。Mustache 支持多种编程语言。

例 2-14 就是将演讲者数据转换成 HTML 内容的 Mustache 模板。

例 2-14　/app/templates/speakers-mustache-template.html

```
<!--
[speakers-mustache-template.html]
此为应用程序首次启动时，演讲者数组数据所使用的模板
-->
<script id="speakerTemplate" type="text/html">
  {{#.}}
    <tr>
      <td>{{firstName}} {{lastName}}</td>
      <td>{{about}}</td>
      <td>{{tags}}</td>
    </tr>
  {{/.}}
</script>
```

以下是对这一示例的一些解释。

- 模板以外部文件的形式存在，将 HTML 与 JavaScript 代码分离。
- 模板代码包含在 `<script>` 元素中。
- 模板中的 HTML 结构与普通网页中的 HTML 结构相同。
- Mustache 使用包含在双括号中的变量来填充数据。
- Mustache 使用参照系来遍历演讲者数组。从 HTTP 调用中返回的是一个匿名集合，因此我们将所有的元素包含在 `{{#.}}` 和 `{{/.}}` 之间来设定参照系。如果返回的数组属于某个字段（如 `speakers`），则参照系会以 `{{#speakers}}` 开头，以 `{{/speakers}}` 结束。
- 模板中的变量表示特定参照对象中的字段名。比如，遍历演讲者数组时，`{{firstName}}` 变量可以获取当前成员的 `firstName` 字段的值。

如需深入了解 Mustache，可参考 Wern Ancheta 撰写的一篇很不错的文章“Easy Templating with Mustache.js”。

除了 Mustache，JavaScript 社区也经常使用其他一些模板库。

Handlebars.js
　Handlebars 和 Mustache 非常相似。

Underscore.js
　Underscore 是一个通用工具库，其中包含一些模板方面的功能。

另外，大多数 MVC 框架（AngularJS、Ember 和 Backbone）都自带某种形式的模板功能。第 7 章将详细介绍 Mustache 和 Handlebars。

例 2-15 展示了重构后使用 Mustache 的 app/scripts/main.js 文件。

例 2-15　speakers-web-3/app/scripts/main.js

```
'use strict';

console.log('Hello JSON at Work!');

$(document).ready(function() {
```

```
  function addSpeakersMustache(speakers) {
    var tbody = $('#speakers-tbody');

    $.get('templates/speakers-mustache-template.html', function(templatePartial) {
      var template = $(templatePartial).filter('#speakerTemplate').html();
      tbody.append(Mustache.render(template, speakers));
    }).fail(function() {
      alert("Error loading Speakers mustache template");
    });

  }

  $.getJSON('http://localhost:5000/speakers',
    function(data) {
      addSpeakersMustache(data);
    }
  );

});
```

在以上示例中，`addSpeakerMustache()` 函数使用之前的 Mustache 模板将 `json-server` 所提供的演讲者数据转换成 HTML。我们使用 jQuery 的 `$.get()` 方法来下载 Mustache 模板文件。下载完成后，使用之前的方法找到主页上的 `<tbody>` 元素，然后使用 `append()` 方法将 `Mustache.rend()` 生成的 HTML 内容添加到该元素中。

最后，我们需要将 Mustache 添加到 Web 应用程序中。

- 使用 Bower 在 Web 应用程序中安装 Mustache。可在命令行中切换到 speakers-web-3 目录，然后运行 `bower install mustache`。
- 在 app/index.html 文件中添加 Mustache（紧邻 main.js），如例 2-16 所示。

例 2-16　speakers-web-3/app/index.html

```
<!doctype html>
<html lang="">

  ...

  <body>

    ...

    <script src="bower_components/mustache.js/mustache.js"></script>

    ...

  </body>
</html>
```

如果之前一直保持应用程序运行，那么可看到如图 2-5 所示的截图。

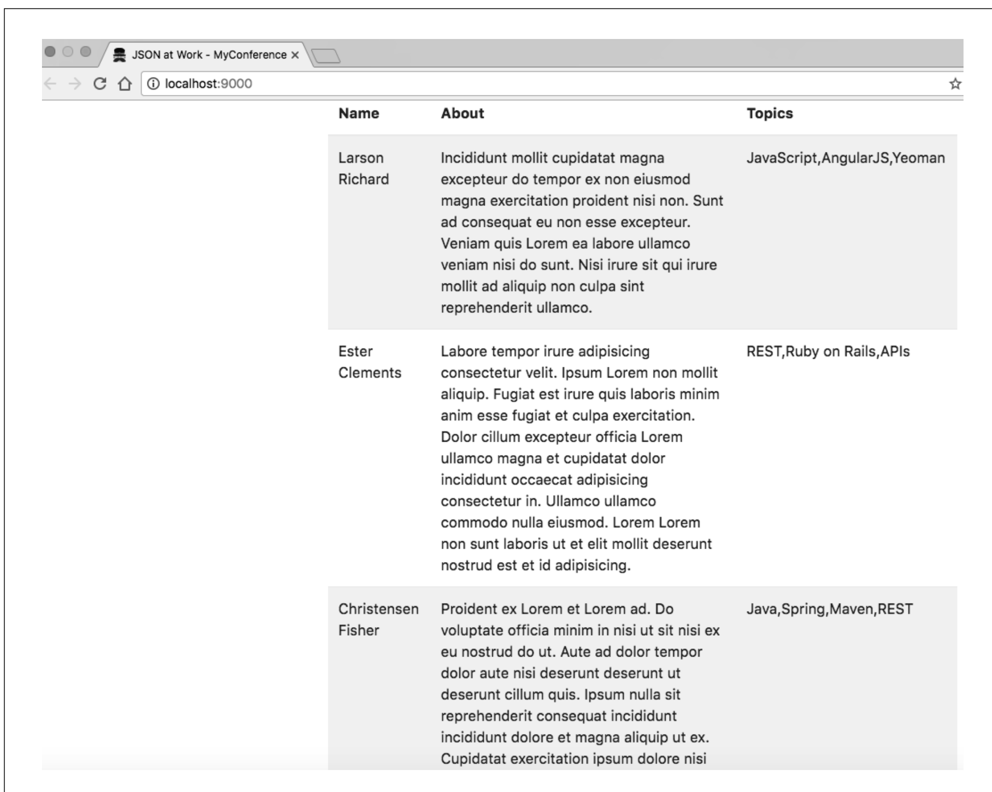


图 2-5：使用 Mustache 来显示演讲者数据

与之前相比，由 Mustache 所显示的演讲者数据存在些许不一致，但因为这种做法通过调用模拟 API 来获取数据，同时使用 Mustache 对 HTML 进行模板处理，所以 Web 应用程序的质量有了进一步的提升。

当然，也可以通过使用 AngularJS 或者 React 来继续深入，这一点还是留给你自己来进行吧。

在继续深入前，请在命令行窗口中使用 Ctrl-C 关闭之前所启动的 Web 应用程序和 `json-server`。

2.6 如何继续深入学习JavaScript

深入、全面地了解 JavaScript 对于真正理解 Node.js 和 JavaScript 框架（如 Angular、React、Ember、Backbone 等）以及 Yeoman 这样的构建管理工具是不可或缺的。如果对 JavaScript 对象感到陌生，觉得各种花括号、圆括号和分号看起来像是语法的大杂烩，别担心，你并非个例。每位 JavaScript 开发者在成长过程中都会碰到类似的问题。

以下是可用于加深并拓展相关技能的网站。

JavaScriptIsSexy 网站提供了非常不错的免费教程来帮助开发者提升至中级或高级水平，主要包括以下 3 个教程：

- 如何正确地学习 JavaScript；
- 学习中高级 JavaScript；
- JavaScript 内行必备技能（Apply、Call 和 Bind）。

随着对这些资料的学习，达到中高级 JavaScript 水平后，对象和函数表达式这些概念就会变成你的常识。此时，使用本章中所提及的 JavaScript 工具和框架就会变得更得心应手、更有效率。

2.7 本章回顾

本章先描述了 JavaScript 和 JSON 间简单的转换工作，然后逐步开发了一个可运行的 Web 应用程序，同时也编写了向 `json-server` 发起 RESTful API 调用的单元测试。为简明起见，本章仅介绍足以理解核心概念的几项技术，并用其搭建了简单的应用程序。至此，我们对 JavaScript、Node.js 和 Yeoman 只是进行了粗浅的讲解。

2.8 内容预告

使用 JavaScript 和 JSON 开发 Web 应用程序后，第 3 章将介绍 JSON 在 Ruby on Rails 中的使用情况。

第 3 章

在Ruby on Rails中使用JSON

上一章介绍了如何在 JavaScript 中使用 JSON，本章将展示 JSON 在 Ruby on Rails 中的使用情况。

本章内容如下所示：

- 在 Ruby 中使用 `MultiJson` 来执行 JSON 的序列化 / 反序列化操作；
- 使用 Ruby 对象和 JSON；
- 理解 JSON 中驼峰式命名的重要性；
- 在 Minitest 中使用 JSON；
- 使用 Minitest 和 `jq` 发起 RESTful API 调用，并对调用结果进行测试；
- 使用 Rails 5 搭建简单的 JSON API。

在本章的应用示例中，我们将对第 1 章中部署到 `json-server` 上的数据进行 RESTful API 调用。然后创建一个更加真实的基于 JSON 的 Web API。开发 RESTful API 之前，我们先来了解一下在 Ruby 和 JSON 之间进行转换的基础知识。

3.1　安装Ruby on Rails

正式开始前，请参考 A.3 节中的操作说明来搭建开发环境。

3.2　Ruby中与JSON有关的gem包

Ruby 中存在多个提供 JSON 序列化 / 反序列化功能的 gem 包，举例如下。

`JSON`

　Ruby 默认提供的 JSON 包。

`oj`

经过优化的 JSON 包，很多人认为 `oj` 是 Ruby 中最快的 JSON 处理工具。

`yajl`

全称为 Yet Another JSON Library，即“另一个 JSON 类库”。

除此之外，还有很多有关 JSON 的 gem 包，因此很难选择。在这种情况下，`MultiJson` 对 gem 包的选择和调用进行了封装，自动选择当前应用程序环境中最快的 JSON 包，从而使得开发者无须学习每个 JSON 包的使用。像这样对 gem 包所进行的封装操作可以在应用程序和某个具体的 JSON 实现之间实现解耦。如需了解更多有关 `MultiJson` 如何选择 JSON 实现的信息，可以参考其 GitHub 主页。如需了解详细文档，可参考 RubyDoc。

默认情况下，`MultiJson` 使用标准的 JSON gem 包，安装 `oj` 后则可对性能进行优化。

```
gem install multi_json
gem install oj
```

安装 `oj` gem 包后，`MultiJson` 将默认使用 `oj`，而不是标准的 JSON。

3.3 用MultiJson进行序列化/反序列化操作

应用程序需要有能力将 Ruby 的数据类型转换为 JSON（序列化），也需要有能力进行相反的操作（反序列化），从而与其他应用程序进行 JSON 数据的交换。

3.3.1 MultiJson对象

`MultiJson` 对象提供以下方法。

- `MultiJson.dump()` 将 Ruby 数据序列化为 JSON。
- `MultiJson.load()` 将 JSON 反序列化为 Ruby 数据。

值得注意的是，`MultiJson.dump()` 会执行以下操作。

- 当使用 `oj` 来序列化 `speaker` 对象时,使用的是 Ruby 中传统的下划线分隔命名法（`first_name`），而不是推荐的更有利于跨平台的驼峰式命名法（`firstName`）。
- 当使用 `JSON` 来序列化 `speaker` 对象时，序列化操作并不会生成 JSON 字符串。这是因为对于 `JSON` gem 包来说，只有某个类实现了 `to_json()` 方法后，才能成功序列化该类的对象实例。
- 对于 JSON 中的键名，使用的是下划线分隔命名法（`first_name`），而不是驼峰式命名法（`firstName`）。

根据 `MultiJson` 的 RubyDoc 文档，以下是 `MultiJson.dump()` 方法的语法：

```
#dump(object, options = {})
```

因为 `MultiJson` 只是一个封装，所以 `options` 的用法取决于具体所采用的 JSON 实现（在本节示例中，此实现即为 `oj`）。

3.3.2 Ruby中简单数据类型的JSON序列化/反序列化操作

首先，我们看一下 Ruby 中基础数据类型的序列化操作：

- 整数型
- 字符串
- 布尔值
- 数组
- 哈希
- 对象

例 3-1 展示了如何使用 `MultiJson` 和 `oj` 来对简单的 Ruby 数据类型进行序列化 / 反序列化操作。

例 3-1 ruby/basic_data_types_serialize.rb

```
require 'multi_json'

puts "Current JSON Engine = #{MultiJson.current_adapter()}"
puts

age = 39 # 整数型
puts "age = #{MultiJson.dump(age)}"
puts

full_name = 'Larson Richard' # 字符串
puts "full_name = #{MultiJson.dump(full_name)}"
puts

reqistered = true # 布尔值
puts "reqistered = #{MultiJson.dump(reqistered)}"
puts

tags = %w(JavaScript, AngularJS, Yeoman) # 字符串数组
puts "tags = #{MultiJson.dump(tags)}"
puts

email = { email: 'larsonrichard@ecratic.com' } # 哈希
puts "email = #{MultiJson.dump(email)}"
puts

class Speaker
  def initialize(first_name, last_name, email, about,
                 company, tags, registered)
    @first_name = first_name
    @last_name = last_name
    @email = email
    @about = about
    @company = company
    @tags = tags
```

```
    @registered = registered
  end
end

speaker = Speaker.new('Larson', 'Richard', 'larsonrichard@ecratic.com',
            'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
            'Ecratic', %w(JavaScript, AngularJS, Yeoman), true)

puts "speaker (using oj gem) = #{MultiJson.dump(speaker)}"
puts
```

在命令行中运行 `ruby basic_data_types_serialize.rb` 可以得到以下结果：

```
json-at-work => ruby basic_data_types_serialize.rb
Current JSON Engine = MultiJson::Adapters::Oj

age = 39

full_name = "Larson Richard"

registered = true

tags = ["JavaScript,","AngularJS,","Yeoman"]

email = {"email":"larsonrichard@ecratic.com"}

speaker (using oj gem) = {"first_name":"Larson","last_name":"Richard","email":"larsonrichard@ecratic.com","about":"Incididunt
 mollit cupidatat magna excepteur do tempor ex non ...","company":"Ecratic","tags":["JavaScript,","AngularJS,","Yeoman"],"reg
istered":true}
```

对于标量类型（整数型、字符串、布尔值）来说，`MultiJson.dump()` 并没有提供什么有意思的功能。但对于 `speaker` 对象来说，`MultiJson.dump()` 可以生成一段驳杂但合法的 JSON 字符串，因此显得比较有用。正如稍后将提到的，`MultiJson.dump()` 还可以接受其他参数，从而使得序列化操作更加强大。

为了增加结果的可读性，我们将使用 `:pretty => true` 选项来优化 `speaker` 对象的 JSON 输出结果，如例 3-2 所示。虽然对输出结果进行优化显示可以增加可读性，但这会降低效率，因此这一选项只应在调试时加以使用。

例 3-2　ruby/obj_serialize_pretty.rb

```
require 'multi_json'

...

speaker = Speaker.new('Larson', 'Richard', 'larsonrichard@ecratic.com',
            'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
            'Ecratic', %w(JavaScript, AngularJS, Yeoman), true)

puts "speaker (using oj gem) = #{MultiJson.dump(speaker, pretty: true)}"
puts
```

运行以上代码可以得到以下优化显示的结果：

```
json-at-work => ruby obj_serialize_pretty.rb
Current JSON Engine = MultiJson::Adapters::Oj

speaker (using oj gem) = {
  "first_name":"Larson",
  "last_name":"Richard",
  "email":"larsonrichard@ecratic.com",
  "about":"Incididunt mollit cupidatat magna excepteur do tempor ex non ...",
  "company":"Ecratic",
  "tags":[
    "JavaScript,",
    "AngularJS,",
    "Yeoman"
  ],
  "registered":true
}
```

3.3.3　用MultiJson进行JSON反序列化操作

除了对数据进行序列化，MultiJson 还可以对 JSON 进行反序列化操作。本节将使用 MultiJson.load() 方法将 JSON 反序列化为 Ruby Hash。因为 speaker 对象的 initialize() 方法接受的参数类型为字符串（与 speaker 对象的属性类型保持一致），所以我们需要对反序列化后的 Hash 进行转换，将其转换为用于初始化 speaker 对象所需的一系列属性。好在知名类库 OpenStruct 可以使得 Hash 表现出对象的行为，因此我们无须编写任何代码即可完成这一转换工作。

例 3-3 展示了 OpenStruct 的使用。

例 3-3　ruby/ostruct_example.rb

```
require 'ostruct'

h = { first_name: 'Fred' }
m = OpenStruct.new(h)
puts m            # 打印结果: #<OpenStruct first_name="Fred">
puts m.first_name  # 打印结果: Fred
```

OpenStruct 是和 Hash 类似的一种数据结构，它允许对属性及其值进行名称 – 值对定义，也提供以属性的形式访问键值的功能。OpenStruct 是 Ruby 核心语法的一部分。如需了解更多有关 OpenStruct 的信息，可参考 Ruby 的核心文档。

当实例化一个新的 speaker 对象时，若能以可读的形式打印出这一新对象，那么对调试工作是非常有利的。如果使用普通的 puts 语句，得到的结果一般如下所示：

```
puts speaker # #<Speaker:0x007f84412e0e38>
```

如果使用 awesome_print gem 包，则可以得到更具可读性的输出结果。如需了解更多相关信息，可参考 awesome_print 的 GitHub 主页。

运行例 3-4 的代码前，请在命令行中安装 awesome_print 包。

```
gem install awesome_print
```

例 3-4　ruby/obj_deserialize.rb

```ruby
require 'multi_json'
require 'ostruct'
require 'awesome_print'

puts "Current JSON Engine = #{MultiJson.current_adapter()}"
puts

class Speaker
  def initialize(first_name, last_name, email, about,
                 company, tags, registered)
    @first_name = first_name
    @last_name = last_name
    @email = email
    @about = about
    @company = company
    @tags = tags
    @registered = registered
  end
end

speaker = Speaker.new('Larson', 'Richard', 'larsonrichard@ecratic.com',
            'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
            'Ecratic', %w(JavaScript, AngularJS, Yeoman), true)

json_speaker = MultiJson.dump(speaker, pretty: true)
puts "speaker (using oj gem) = #{MultiJson.dump(speaker)}"
puts

ostruct_spkr = OpenStruct.new(MultiJson.load(json_speaker))

speaker2 =  Speaker.new(ostruct_spkr.first_name, ostruct_spkr.last_name,
                     ostruct_spkr.email, ostruct_spkr.about, ostruct_spkr.company,
                     ostruct_spkr.tags, ostruct_spkr.registered)

puts "speaker 2 after MultiJson.load()"
ap speaker2
puts
```

运行这一示例后可以看到，代码成功地将保存在 `json_speaker` 变量中的 JSON 字符串反序列化为 `OpenStruct` 对象，然后再将该 `OpenStruct` 对象转换为新的 `speaker` 实例，即 `speaker2`。注意该示例对 `awesome_print` 中的 `ap` 方法的使用，代码使用了 `ap` 而不是内置的 `puts`，从而优化显示了对象的输出结果。

```
json-at-work => ruby obj_deserialize.rb
Current JSON Engine = MultiJson::Adapters::Oj

speaker (using oj gem) = {"first_name":"Larson","last_name":"Richard","email":"larsonrichard@ecratic.com","about":"Incididunt
 mollit cupidatat magna excepteur do tempor ex non ...","company":"Ecratic","tags":["JavaScript,","AngularJS,","Yeoman"],"reg
istered":true}

speaker 2 after MultiJson.load()
#<Speaker:0x007fc77482a260 @first_name="Larson", @last_name="Richard", @email="larsonrichard@ecratic.com", @about="Incididunt
 mollit cupidatat magna excepteur do tempor ex non ...", @company="Ecratic", @tags=["JavaScript,", "AngularJS,", "Yeoman"], @
registered=true>
```

尽管 `multi_json` 和 `oj` 可以高效地处理 JSON，但有时开发者需要对数据的序列化过程有更多的把控。

3.3.4　关于JSON和驼峰式命名

不管你是否已经注意到，JSON 中的键名 / 属性名通常都是以驼峰式命名形式存在的。比如，表示某人名字的键名一般会命名为 `firstName`。但到目前为止，我们所了解的 Ruby 中的 JSON 类库都是将键名以下划线分隔的形式表示的（`first_name`）。这种做法在自娱自乐的小型代码示例和单元测试中**可能**没有问题，但并不兼容主流做法。以下是具体原因。

- JSON 必须具备互操作性。虽然以下观点可能会冒犯很多热爱 Ruby 的人，也可能会被批评为**无意义的繁文末节**，但我还是不得不说：JSON 和 REST 的全部意义就在于在多种多样的应用程序系统之间实现互操作性。除了 Ruby，还存在其他编程语言，而这些编程语言都是采用驼峰式命名法（`firstName`）的。如果一个 API 以一种意想不到的方式工作，那么是不会有人愿意使用这样的 API 的。
- 在 JSON 中存在使用驼峰式命名法的重量级参与者。
  - Google 的 JSON 编码规范对驼峰式命名法进行了标准化。
  - 基于 JSON 的大多数开放 API 都使用了驼峰式命名法，如亚马逊的 AWS、Facebook 以及 LinkedIn。
- 应该避免单个平台影响整体系统。不论生成或读取 JSON 的平台 / 语言是什么，JSON 的呈现都应该保持不变。Ruby on Rails 社区偏好使用以下划线分隔来命名的做法，这在 Ruby 平台内部没有任何问题，但这一编程语言的内部习惯不应该影响整体 API。

3.3.5　用ActiveSupport进行JSON序列化操作

ActiveSupport 这一 gem 包提供了从 Rails 中抽取出来的一些功能，包括时区、国际化以及 JSON 编码 / 解码等。ActiveSupport 中的 JSON 模块提供了以下功能：

- 在驼峰式命名和下划线分隔命名之间进行转换；
- 选择目标对象的部分内容进行序列化。

可以在命令行中运行以下命令来安装 ActiveSupport：

```
gem install activesupport
```

我们使用 `ActiveSupport::JSON.encode()` 将 `speaker` 对象序列化为 JSON，如例 3-5 所示。

例 3-5　ruby/obj_serialize_active_support.rb

```
require 'active_support/json'
require 'active_support/core_ext/string'

...

speaker = Speaker.new('Larson', 'Richard', 'larsonrichard@ecratic.com',
            'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
            'Ecratic', %w(JavaScript, AngularJS, Yeoman), true)

json = ActiveSupport::JSON.encode(speaker).camelize(first_letter = :lower)
```

```
puts "Speaker as camel-cased JSON \n#{json}"
puts

json = ActiveSupport::JSON.encode(speaker,
                       only: ['first_name', 'last_name'])
                       .camelize(first_letter = :lower)

puts "Speaker as camel-cased JSON with only firstName and lastName \n#{json}"
puts
```

可以看到 ActiveSupport::JSON.encode() 在以上示例中提供了以下选项。

- 通过链式调用 camelize() 将键名转换成驼峰式命名（firstName）。值得注意的是，键名中的首字母默认为大写,因此需要使用 first_letter = :lower 参数将其转换为小写（小驼峰命名法）。
- 使用 only: 参数选择 speaker 对象的一部分进行序列化。

运行以上代码后可以看到以下结果：

```
json-at-work => ruby obj_serialize_active_support.rb
Speaker as camel-cased JSON
{"firstName":"Larson","lastName":"Richard","email":"larsonrichard@ecratic.com","about":"Incididunt mollit cupidatat magna exc
epteur do tempor ex non ...","company":"Ecratic","tags":["JavaScript,","AngularJS,","Yeoman"],"registered":true}

Speaker as camel-cased JSON with only firstName and lastName
{"firstName":"Larson","lastName":"Richard"}
```

如果只是需要将键名从下划线分隔转换为驼峰式命名，那么还可以选择 awrence 包这一简单的替代方案。使用 awrence 将 Hash 键中的下划线分隔命名转换为驼峰式命名后，即可将其转换为驼峰式命名格式的 JSON。对于 awrence 包的具体使用，我的经验不多，因此这一点还是留给你自己来实践吧。

3.3.6 用ActiveSupport进行JSON反序列化操作

ActiveSupport 还可以对 JSON 进行反序列化操作。本节会使用 decode() 方法将 JSON 反序列化为 Hash。和之前一样，我们使用 OpenStruct 和 awesome_print 来辅助创建对象实例，并优化显示结果，如例 3-6 所示。

例 3-6 ruby/obj_deserialize_active_support.rb

```
require 'multi_json'
require 'active_support/json'
require 'active_support/core_ext/string'
require 'ostruct'
require 'awesome_print'

...

speaker = Speaker.new('Larson', 'Richard', 'larsonrichard@ecratic.com',
            'Incididunt mollit cupidatat magna excepteur do tempor ex non ...',
            'Ecratic', %w(JavaScript, AngularJS, Yeoman), true)

json_speaker = ActiveSupport::JSON.encode(speaker)
puts "speaker (using oj gem) = #{ActiveSupport::JSON.encode(speaker)}"
puts ostruct_spkr = OpenStruct.new(ActiveSupport::JSON.decode(json_speaker))
```

```
speaker2 =  Speaker.new(ostruct_spkr.first_name, ostruct_spkr.last_name,
                        ostruct_spkr.email, ostruct_spkr.about, ostruct_spkr.company,
                        ostruct_spkr.tags, ostruct_spkr.registered)

puts "speaker 2 after ActiveSupport::JSON.decode()"
ap speaker2
puts
```

在命令行中运行以上代码后可得到以下结果：

```
json-at-work => ruby obj_deserialize_active_support.rb
speaker (using oj gem) = {"first_name":"Larson","last_name":"Richard","email":"larsonrichard@ecratic.com","about":"Incididunt
 mollit cupidatat magna excepteur do tempor ex non ...","company":"Ecratic","tags":["JavaScript,","AngularJS,","Yeoman"],"reg
istered":true}

speaker 2 after ActiveSupport::JSON.decode()
#<Speaker:0x007fe73c0a4eb8 @first_name="Larson", @last_name="Richard", @email="larsonrichard@ecratic.com", @about="Incididunt
 mollit cupidatat magna excepteur do tempor ex non ...", @company="Ecratic", @tags=["JavaScript,", "AngularJS,", "Yeoman"], @
registered=true>
```

plissken 这一 gem 包可以将 `Hash` 键名中的驼峰式命名转换为下划线分隔命名。接下来的单元测试中将使用 `plissken`。

3.4 用模拟API进行单元测试

了解了如何对 `speaker` 对象进行序列化 / 反序列化操作后，我们可以运行一个简单的服务器端单元测试程序，以对 `json-server` 提供的模拟 API 进行测试。

3.4.1 使用Minitest即可完成单元测试

Minitest（Ruby 核心的一部分）和 RSpec 是最常用的两个 Ruby 测试框架。两者都非常优秀，但为了重点关注 JSON，我们只讨论其中一个框架的使用。

一方面，Minitest：

- 是 Ruby 标准库的一部分，因此无须额外的安装工作；
- 属于轻量级类库，且简单易用；
- 提供了 RSpec 所具备的**大多数**功能。

另一方面，RSpec：

- 需要额外安装 `rspec` 这一 gem 包，不过该 gem 包在 Ruby on Rails 社区中有着广泛的使用；
- 庞大、复杂，其代码库比 Minitest 大 8 倍；
- 匹配规则比 Minitest 更丰富。

两者间的选择更多取决于个人偏好，因为无论哪个测试框架都是可以正常完成工作的。由于 Minitest 是 Ruby 标准库的一部分，因此本书选择使用 Minitest。

Minitest 允许使用 BDD 风格（`Minitest::Spec`）和 TDD 风格（`Minitest::Test`）来编写测试。出于以下原因，我们使用 `Minitest:Spec`。

- 我个人偏好使用 BDD，即用简单的英文语句来描述每个测试用例。
- 这一做法与 RSpec 类似，因此会让使用 RSpec 的开发者感到熟悉。

- 与本书其他章节中基于 JavaScript 的 Mocha/Chai 测试代码保持一致。

本章只介绍了 Minitest 的一些基础知识。如需了解更多信息，可参考 Chris Kottom 所著的 *The Minitest Cookbook*。

3.4.2 设置单元测试环境

正式开始前，需要先完成测试环境的设置工作。如尚未安装 Ruby on Rails，可参考 A.3 节和 A.3.3 节中的内容。如需依照本节的描述运行代码示例中的 Ruby 项目，可使用 `cd` 命令切换到 chapter-3/speakers-test 目录，并执行以下命令来安装项目依赖：

```
bundle install
```

对于 Ruby 项目而言，Bundler 可以帮助进行依赖管理。

如需手动创建本节中的 Ruby 项目，可参考本书在 GitHub 上的相关指导步骤。

3.4.3 测试数据

本节将使用之前章节中所提供的演讲者数据作为测试数据，并将其部署为 RESTful API。与之前的做法相同，我们将使用 `json-server` 这一 Node.js 模块把 data/speakers.json 文件部署为 Web API。如需了解有关安装 `json-server` 的信息，请参考 A.2.5 节中的内容。

以下是在本地机器中使用 5000 端口运行 `json-server` 的步骤：

```
cd chapter-3/data

json-server -p 5000 ./speakers.json
```

访问该 API 时，还可以通过在 URI（如 http://localhost:5000/speakers/1）中添加 `id` 来获取单个演讲者的数据。准备好模拟 API 后，就可以开始编写单元测试了。

3.4.4 用Minitest测试API所提供的JSON

本节的单元测试实现了以下功能：

- 向模拟的演讲者数据 API 发起 HTTP 调用；
- 检查 HTTP 响应体中的值，并将其与预期结果进行对比。

与之前的章节一样，本节会继续使用 Unirest 这一 API 封装工具，但在选择具体类库时会使用 Unirest 的 Ruby 实现。需要注意的是，Unirest 包会接受 HTTP 响应体中的 JSON，将其解析为 Ruby 中的 `Hash`，然后再将该 `Hash` 以 HTTP 响应体的形式返回给调用方。这种做法意味着单元测试用例所测试的并不是原始的 JSON 数据，而是从 API 所提供的 JSON 响应中转换得到的 `Hash`。

3.4.5 对演讲者数据的单元测试

例 3-7 中的单元测试展示了如何使用 Unirest 向 `json-server` 所提供的模拟 API 发起调用，并测试响应。

例 3-7　speakers-test/test/speakers_spec.rb

```
require 'minitest_helper'

require 'unirest'
require 'awesome_print'
require 'ostruct'
require 'plissken'
require 'jq/extend'

require_relative '../models/speaker'

describe 'Speakers API' do
  SPEAKERS_ALL_URI = 'http://localhost:5000/speakers'

  before do
    @res = Unirest.get SPEAKERS_ALL_URI,
                    headers:{ 'Accept' => "application/json" }

  end

  it 'should return a 200 response' do
    expect(@res.code).must_equal 200
    expect(@res.headers[:content_type]).must_equal 'application/json; charset=utf-8'
  end

  it 'should return all speakers' do
    speakers = @res.body
    expect(speakers).wont_be_nil
    expect(speakers).wont_be_empty
    expect(speakers.length).must_equal 3
  end

  it 'should validate the 3rd speaker as an Object' do
    speakers = @res.body
    ostruct_spkr3 = OpenStruct.new(speakers[2].to_snake_keys())

    expect(ostruct_spkr3.company).must_equal 'Talkola'
    expect(ostruct_spkr3.first_name).must_equal 'Christensen'
    expect(ostruct_spkr3.last_name).must_equal 'Fisher'
    expect(ostruct_spkr3.tags).must_equal ['Java', 'Spring', 'Maven', 'REST']

    speaker3 =  Speaker.new(ostruct_spkr3.first_name, ostruct_spkr3.last_name,
                            ostruct_spkr3.email, ostruct_spkr3.about,
                            ostruct_spkr3.company, ostruct_spkr3.tags,
                            ostruct_spkr3.registered)

    expect(speaker3.company).must_equal 'Talkola'
    expect(speaker3.first_name).must_equal 'Christensen'
    expect(speaker3.last_name).must_equal 'Fisher'
    expect(speaker3.tags).must_equal ['Java', 'Spring', 'Maven', 'REST']
  end
```

```
  it 'should validate the 3rd speaker with jq' do
    speakers = @res.body
    speaker3 = speakers[2]

    speaker3.jq('.company') {|value| expect(value).must_equal 'Talkola'}
    speaker3.jq('.tags') {|value|
        expect(value).must_equal ['Java', 'Spring', 'Maven', 'REST']}
    speaker3.jq('.email') {|value|
        expect(value).must_equal 'christensenfisher@talkola.com'}
    speaker3.jq('. | "\(.firstName) \(.lastName)"') {|value|
        expect(value).must_equal 'Christensen Fisher'}
  end

end
```

对于这一单元测试，需要注意以下几点。

- `minitest_helper` 将配置工作进行归总，并将其从测试代码中提取出来。本章稍后将详细介绍 Minitest 中的辅助模块。
- 测试代码在 Minitest 的 `before` 方法中同步地执行 Unirest 的 `GET` 请求，并获得响应，从而避免配置代码的重复。在 `describe` 语句所定义的范围内，Minitest 会在执行每个测试用例（即 `it` 语句）前运行一次 `before()` 方法。
- `should return all speakers` 这一测试用例执行了以下操作。
  – 确认 HTTP 响应体不为空。
  – 检查 API 所返回的演讲者数目是否为 3。
- `should validate the 3rd speaker as an Object` 这一测试用例执行了以下操作。
  – 从 HTTP 响应体（`@res.body`）中抽取出以 `Hash` 形式表示的演讲者数据。此时，Unirest 已经对由 API 所提供的 JSON 进行了解析，并将其转换成了 `Hash`。
  – 使用 `OpenStruct.new()` 将第三个演讲者的 `Hash` 数据转换为 `OpenStruct` 结构。`plissken` gem 提供的 `to_snake_keys()` 方法可以将 `Hash` 键名中的驼峰式命名（`firstName`）转换成以下划线分隔命名（`first_name`），从而与 Ruby 中的编程习惯保持兼容。
  – 使用 Minitest 中 BDD 风格的 `expect` 断言语句来检查期望结果。
    ♦ 第三个演讲者的 `company`、`first_name`、`last_name` 和 `tag` 值需要与 speakers.json 文件中的值保持一致。
- `should validate the 3rd speaker with jq` 这一测试用例的工作机制如下所示。
  – 使用 `jq` 查询语句（如 `.company`）来检查测试中的字段值。`jq` 允许对基于 JSON 的 `Hash` 进行直接查询，而无须将 `Hash` 转换为对象，因此能简化单元测试的编写工作。`jq` 是一个强大的 JSON 搜索工具，第 6 章将对其进行详细介绍。
  – `. | "\(.firstName) \(.lastName)"` 查询语句对 `firstName` 和 `lastName` 字段的值进行字符串拼接，以便得到演讲者的全名后再进行相关测试。
  – `ruby-jq` gem 提供了 `jq` 在 Ruby 中的实现。

在命令行中使用 `bundle exec rake` 来执行该测试，可以得到以下结果：

```
json-at-work => bundle exec rake
Started with run options --seed 42108

Speakers API
  test_0001_should return a 200 response                    PASS (0.01s)
  test_0004_should validate the 3rd speaker with jq         PASS (0.02s)
  test_0003_should validate the 3rd speaker as an Object    PASS (0.00s)
  test_0002_should return all speakers                      PASS (0.00s)

Finished in 0.03299s
4 tests, 18 assertions, 0 failures, 0 errors, 0 skips
```

`rake` 是 Ruby 中常用的构建工具。`bundle exec rake` 命令执行了以下具体操作。

- `rake` 使用了项目的 Gemfile 中所列举的 gem 包。
- `rake` 配置中将 `test` 作为默认任务。

Rakefile 文件中定义了具体的构建任务，如例 3-8 所示。

例 3-8　speakers-test/Rakefile

```ruby
require 'rake/testtask'

Rake::TestTask.new(:test) do |t|
  t.libs = %w(lib test)
  t.pattern = 'test/**/*_spec.rb'
  t.warning = false
end

task :default => :test
```

默认情况下，Minitest 不会显示测试是否已通过。而在之前的单元测试运行结果中，我们看到输出结果中显示了通过测试的用例。这是因为 speakers-test 项目中使用了 `minitest-reporters` gem 来提升输出结果的可读性。

例 3-9 中的 Minitest 辅助模块对 `speakers_spec` 中所使用的 `minitest` 和 `minitest-reporters` gem 进行了配置。

例 3-9　speakers-test/test/minitest_helper.rb

```ruby
require 'minitest/spec'
require 'minitest/autorun'

require "minitest/reporters"
Minitest::Reporters.use! Minitest::Reporters::SpecReporter.new
```

考虑到完整性，例 3-10 展示了保存演讲者数据的 `Speaker` 类。

例 3-10　speakers-test/models/speaker.rb

```ruby
class Speaker
  attr_accessor :first_name, :last_name, :email,
                :about, :company, :tags, :registered

  def initialize(first_name, last_name, email, about,
                 company, tags, registered)
    @first_name = first_name
    @last_name = last_name
    @email = email
```

```
    @about = about
    @company = company
    @tags = tags
    @registered = registered
  end
end
```

这段代码很普通也很简单。

- 为遵循 Ruby 项目中普遍接受的惯例，speaker.rb 文件保存在 models 文件夹中。
- `attr_accessor` 定义了 `Speaker` 类的数据成员(如 `first_name`)及其 getter/setter 访问方法。
- 当调用 `Speaker.new()` 时，`initialize()` 方法负责对数据成员进行初始化。

继续阅读前，请在命令行中敲击 Ctrl-C 来关闭 `json-server`。

3.4.6 有关Ruby和Minitest的更多学习资料

本章只介绍了 Ruby 和 Minitest 的基础知识。如需继续深入学习，可参考以下学习资料：

- Jeremy McAnally 和 Assaf Arkin 所著的《Ruby 程序员修炼之道（第 2 版）》[1]；
- David A. Black 所著的 *The Well-Grounded Rubyist, 2nd Ed.*（Manning 出版社）；
- Chris Kottam 所著的 *Minitest Cookbook*。

3.4.7 似乎少了点什么

到目前为止，单元测试成功地对 JSON 数据进行了检测，测试过程还算优雅，但并不完美。原因在于测试代码必须一一检查所有的预期字段，这无疑是笨拙而繁琐的。对于庞大而又结构复杂的 JSON 文档来说，可以想象同样的做法将会带来何等艰苦的工作。对于这一问题，其中一个解决方案是使用 JSON Schema。第 5 章将对此进行详细介绍。

本节展示了如何部署和调用模拟 API，接下来我们将用 Ruby on Rails 搭建一个小型的 RESTful API。

3.5 用Ruby on Rails搭建小型Web API

了解了如何对 `speaker` 对象进行序列化 / 反序列化操作，以及如何对 `json-server` 提供的模拟 API 进行单元测试后，本节将介绍如何使用 API 数据来搭建简单的 Web 应用程序，并将其展示给用户。

我们将继续使用演讲者数据，并通过 Rails 5 将其创建为 API。因为 Rails 5 这一版本包含了 `rails-api`，所以可用于创建一个仅包含 API 功能的 Rails 应用程序。`rails-api` 一开始是一个独立的 gem，之后才被整合为 Rails 框架的一部分。

本节将搭建 2 个基于 Rails 的 API 应用程序，以演示 AMS 中的一些相关功能。

speakers-api-1
　创建 API，提供驼峰式命名格式的 JSON。

注 1：此书已由人民邮电出版社出版。——编者注

speakers-api-2
: 创建 API，提供定制化的 JSON 文档。

正式开始前，我们先来看一下 API 中 JSON 生成工具的选择。

3.5.1 选择JSON序列化工具

Ruby on Rails 中存在多个 JSON 生成方案。以下列举了一些使用最广泛的技术。

ActiveModel::Serializers (AMS)
: AMS 为对象提供序列化、校验等功能。它是 Rails API 的一部分，可在其 GitHub 主页上找到相关详细文档。

Jbuilder
: Jbuilder 是一个领域专用语言（Domain-Specific Language，DSL）构建器，可以读取模板文件并对最终的输出进行相应的控制。有关 Jbuilder 的详细信息，可以参考其 GitHub 主页。

RABL
: RABL（Ruby API Builder Language，Ruby API 构建器语言）可生成 JSON、XML、PList、MessagePack 和 BSON。这一 gem 也使用了模板文件。有关 RABL 的详细信息，可以参考其 GitHub 主页。

1. 评估标准

以下是选择 JSON 序列化方案时的一些考虑点。

- JSON 的生成操作应当与操作的目标对象隔离，因为目标对象自身不应该获知其具体的外界表现形式。这意味着目标对象中不应该存在任何与 JSON 生成有关的代码。根据 Bob Martin 大叔的说法，如果需要修改一个类，那么只应该存在一个修改的理由，即单一职责原则（面向对象设计中 SOLID 原则中的第一个原则）。如需了解细节信息，可参考 Bob Martin 的面向对象设计原则网站。当将 JSON 格式化工作引入对象内部时，因为担负了以下两个职责，所以该对象就存在第二个修改理由了（而这无疑会加重未来代码修改的难度）。
  - 对象原有的功能。
  - JSON 编码功能。
- 控制类和模型类不应该与 JSON 的生成操作混杂在一起。如果发生混杂，同样会违反单一职责原则，并使得控制类 / 模型类的代码变得不那么灵活。正确的做法应当是使用外部模板来隔离散乱、复杂的格式化逻辑，以使控制类和模型类保持干净的状态。
- 应当可以选择对象的部分属性进行序列化，或者在序列化时忽略部分属性。

上述考虑点看起来有些失之过严，但这么做的目的在于确保互操作性和一致性。不过，因为没有银弹[2]，所以出现别的意见也完全在意料之中。当考虑别的做法时，以下建议值得参考。

注 2：在欧洲民间传说及 19 世纪以来哥特小说风潮的影响下，银色子弹往往被描绘成具有驱魔功效的武器，后比喻为具有极端有效性的解决方法。——译者注

- 明白自己做出决策的原因。物色可靠的软件工程师和架构师来作为自己的后备力量。
- 与其他人合作并打成一片。确认你所采用的做法与整个技术社区兼容，而不是仅适用于单门语言、单个平台或者单个部门这样的特定技术领域。

建立评估标准后，我们来看一下相应的选项。

2. AMS、RABL还是Jbuilder

基于之前描述的考虑点，并回顾所有的选项后，我们发现这一决策比较棘手，因为 AMS、RABL 和 Jbuilder 都能够提供我们所需要的绝大多数特性。AMS 将序列化操作提取到一个单独的 `Serializer` 对象中，RABL 和 Jbuilder 则使用了外部模板。因为 RABL 无法生成小驼峰式命名的 JSON，所以被排除在外，这样选项就只剩下 AMS 和 Jbuilder 了。

在 AMS 和 Jbuilder 之间作出选择是比较困难的。

- 两者提供的 JSON 质量差不多。
- 当配置 Rails 来使用 `oj` 时，两者的性能没有差别。

最终的决策取决于偏好。

- 使用 `Serializer` 对象程序化地执行 JSON 序列化操作（AMS）还是使用模板技术（Jbuilder）。
- JSON 序列化操作发生在控制器中（AMS）还是发生在表现层中（Jbuilder）。

双方都拥有不少支持的观点。

支持 AMS 的观点
: AMS 中所有的代码都是 Ruby 代码，而 Jbuilder 对模板技术的使用要求开发者学习一门新的领域专用语言，因此 AMS 方案更佳。

支持 Jbuilder 的观点
: 使用 Jbuilder 会迫使开发者优先考虑 JSON 的呈现，并推动 JSON 与底层数据库之间的解耦。

正如 Rails 社区中很多人所说的那样，在 AMS 和 Jbuilder 之间作出选择是非常困难的，两者间没有明显的优劣；不管怎么选择，都可以在 API 中生成高质量的 JSON 响应。我选择 AMS，因为它是 Rails 自带的，并且使用 AMS 无须再学习一门新的领域专用语言。

3.5.2 speakers-api-1——创建API以提供驼峰式命名风格的JSON

本节将使用 Rails 5，通过执行以下步骤来创建并部署 speakers-api-1 这一 API。

(1) 建立项目。

(2) 编写源代码。

  - 模型
  - 序列化工具类
  - 控制器

(3) 部署 API。

(4) 用 Postman 进行测试。

1. 建立speakers-api-1项目

speakers-api-1 项目已经包含在本章的代码示例中，具体位于 chapter-3/speakers-api-1 目录下，因此无须从头开始创建该项目。但为了保持完整性，以下边栏描述了如何创建该项目。

用 Rails 创建 speakers-api-1 应用程序

可使用以下命令创建 speakers-api-1 这一 Rails API 项目：

```
rails new speakers-api-1 -T --api --skip-active-record --skip-action-mailer
--skip-action-cable
```

在这一示例中，我们不需要 Rails 通常所提供的前端文件（ERB、JS、CSS、Asset Pipeline 等），也不需要数据库。上述命令所创建的 Rails API 应用程序不包含以下内容。

- 基于 Web 的前端文件。`--api` 选项移除了以下资源。
  - Asset Pipeline
  - 表现层文件
- 测试（使用 `-T` 选项进行移除）。
- ActiveRecord（使用 `--skip-active-record` 选项进行移除）。这意味着应用程序无须数据库即可运行。虽然听起来有些奇怪，但这么做能减少应用程序的外部依赖、简化项目创建过程，因此符合我们的目标。
- ActionMailer（使用 `--skip-action-mailer` 选项进行移除）。我们创建的 Web API 不需要发送电子邮件。
- ActionCable（使用 `--skip-action-cable` 选项进行移除）。我们创建的 API 不使用 WebSocket。

上述命令中的 Rails 生成工具依旧会创建控制器，本节稍后将对此进行介绍。

`rails new` 命令会创建 speakers-api-1 目录。

为了在项目中安装并使用 AMS，示例代码的 Gemfile 文件中添加了以下两行：

```
gem 'active_model_serializers'
gem 'oj'
```

如本章之前所示，我们继续使用 `oj` 来提升性能，但这对于 AMS 来说并不是必需的。

完成项目的创建后，需要安装相关的 gem 来启动项目。安装操作如下所示：

```
cd speakers-api-1

bundle exec spring binstub --all
```

在这一命令中，Bundler 会根据项目的 Gemfile 文件来安装指定的 gem。

2. 创建模型

例 3-11 中的 `Speaker` 类是一个普通的 PORO，用于表示演讲者数据，API 则会将这些数据以 JSON 的形式呈现出来。

例 3-11　speakers-api-1/app/models/speaker.rb

```
class Speaker < ActiveModelSerializers::Model
  attr_accessor :first_name, :last_name, :email,
                :about, :company, :tags, :registered

  def initialize(first_name, last_name, email, about,
                 company, tags, registered)
    @first_name = first_name
    @last_name = last_name
    @email = email
    @about = about
    @company = company
    @tags = tags
    @registered = registered
  end
end
```

除了为 `speaker` 对象实例提供数据成员、构造器和 getter/setter 访问方法外，以上代码没有承担太多工作。代码中也没有包含 JSON 格式方面的任何信息。`Speaker` 类继承自 `ActiveModelSerializers::Model`，因此 AMS 可以将其转换为 JSON。

3. 创建序列化工具类

AMS 提供了独立于控制器和模型的序列化工具，用于将对象序列化为 JSON。本章的代码示例中已经包含了 `SpeakerSerializer` 类，以下边栏描述了该类的创建过程。

生成 SpeakerSerializer

可以在 speakers-web-1 目录下执行以下命令来生成 `speaker` 模型所对应的 `SpeakerSerializer` 类：

```
bin/rails generate serializer speaker
```

这一命令会生成一个仅包含 `id` 字段的序列化工具类：

```
class SpeakerSerializer < ActiveModel::Serializer
  attributes :id
end
```

以此为基础，可以添加用于 JSON 序列化的字段。

例 3-12 展示了在 AMS 中用于将 `speaker` 对象输出为 JSON 的 `SpeakerSerializer` 类。

例 3-12　speakers-api-1/app/models/speaker_serializer.rb

```
class SpeakerSerializer < ActiveModel::Serializer
  attributes :first_name, :last_name, :email,
             :about, :company, :tags, :registered
end
```

在这一示例中，`attributes` 列举了会序列化为 JSON 的所有字段。

4. 创建控制器

在 Rails 应用程序中，控制器负责接受 HTTP 请求并返回 HTTP 响应。在本节示例中，演讲者数据的 JSON 会以 HTTP 响应体的形式返回。本章的代码示例中已经包含了 `SpeakersController` 类，以下边栏描述了该类的创建过程。

> **生成 SpeakersController**
>
> 可以在 speakers-web-1 目录下执行以下命令来生成 `SpeakersController` 类：
>
> ```
> bin/rails generate controller speakers index show
> ```
>
> 这一命令会生成一个包含空白的 `index` 和 `show` 方法的控制类，同时在 app/config/routers.rb 文件中添加相应的 HTTP 路由。

例 3-13 展示了实现 `index` 和 `show` 方法的完整 `SpeakersController` 类。

例 3-13 speakers-api-1/app/controllers/speakers_controller.rb

```
require 'speaker'

class SpeakersController < ApplicationController
  before_action :set_speakers, only: [:index, :show]

  # GET /speakers
  def index
    render json: @speakers
  end

  # GET /speakers/:id
  def show
    id = params[:id].to_i - 1

    if id >= 0 && id < @speakers.length
      render json: @speakers[id]
    else
      render plain: '404 Not found', status: 404
    end
  end

  private

  def set_speakers
    @speakers = []

    @speakers << Speaker.new('Larson', 'Richard', 'larsonrichard@ecratic.com',
      'Incididunt mollit cupidatat magna ...', 'Ecratic',
      ['JavaScript', 'AngularJS', 'Yeoman'], true)

    @speakers << Speaker.new('Ester', 'Clements', 'esterclements@acusage.com',
      'Labore tempor irure adipisicing consectetur ...', 'Acusage',
      ['REST', 'Ruby on Rails', 'APIs'], true)

    @speakers << Speaker.new('Christensen', 'Fisher',
```

```
        'christensenfisher@talkola.com', 'Proident ex Lorem et Lorem ad ...',
        'Talkola',
        ['Java', 'Spring', 'Maven', 'REST'], true)
  end

end
```

关于这段代码，以下几点值得一提。

- `speakers` 数组是硬编码的，仅适用于测试。真实的应用程序中会有一个单独的数据层来负责从数据库中读取 `speakers` 信息，或者通过外部 API 调用来获取 `speakers` 数据。
- `index` 方法执行了以下操作。
  - 响应 /speakers URI 上的 HTTP `GET` 请求。
  - 读取整个 `speakers` 数组，将其转换成 JSON 数组的格式后以 HTTP 响应体的形式输出。
- `show` 方法执行了以下操作。
  - 响应 /speakers/{id} URI（`id` 表示演讲者的 ID）上的 HTTP `GET` 请求。
  - 根据演讲者的 ID 读取 `speaker` 对象数据，将其转换成 JSON 对象的格式后以 HTTP 响应体的形式输出。
  - 如果 HTTP 请求中的 `id` 值越界，则控制器会返回 HTTP 404 状态码（资源不存在），同时使用 `render plain` 语句在 HTTP 响应中包含一段文本信息。
- 当控制器调用 `render` 方法时，Rails 会寻找匹配的序列化工具对 `speaker` 对象进行序列化，默认情况下使用的工具即为 `SpeakerSerializer` 类。

控制器和序列化工具之间是解耦的，彼此不会意识到对方的存在。执行序列化操作的具体代码存在于序列化工具中，不会出现在控制器或者模型中。控制器、模型和序列化工具三者各司其职。

在 Rails 应用程序中，Routes 文件负责在 URL 和相应的控制器方法间实现映射。前面示例中的 `rails generate controller` 命令可以用来创建路由，如例 3-14 所示。

例 3-14　speakers-api-1/app/config/routes.rb

```
Rails.application.routes.draw do
  get 'speakers/index'

  get 'speakers/show'

  # 有关本文件中DSL的详细信息，可查看：
  # http://guides.rubyonrails.org/routing.html
end
```

可以通过使用资源路由来减少 Routes 文件的代码量，如例 3-15 所示。

例 3-15　speakers-api-1/app/config/routes.rb

```
Rails.application.routes.draw do

  resources :speakers, :only => [:show, :index]

  # 有关本文件中DSL的详细信息，可查看：
  # http://guides.rubyonrails.org/routing.html
end
```

与单独定义 index 和 show 方法的路由不同，资源路由这一形式仅使用一行代码就完成了对路由的定义。

5. 让AMS的JSON输出采用驼峰式命名格式

默认情况下，AMS 输出的 JSON 键名会采用下划线分隔的格式（first_name 和 last_name）。从应用程序外部来看，当向 http://localhost:3000/speakers/1 发送 HTTP GET 请求时，得到的 JSON 序列化结果为：

```
{
  "first_name": "Larson",
  "last_name": "Richard",
  "email": "larsonrichard@ecratic.com",
  "about": "Incididunt mollit cupidatat magna ...",
  "company": "Ecratic",
  "tags": [
    "JavaScript",
    "AngularJS",
    "Yeoman"
  ],
  "registered": true
}
```

为了使得 JSON 的输出结果与非 Ruby 客户端兼容，可以添加全局初始化文件将输出格式转为驼峰式命名，如例 3-16 所示。

例 3-16　speakers-api-1/config/initializers/active_model_serializers.rb

```
ActiveModelSerializers.config.key_transform = :camel_lower
```

6. 部署API

在 speakers-api-1 目录中运行 rails s 命令，将 API 部署到 http://localhost:3000/speakers 这一 URL 上，可以在命令行窗口中看到以下结果：

```
json-at-work => rails s
=> Booting Puma
=> Rails 5.0.2 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.8.2 (ruby 2.4.0-p0), codename: Sassy Salamander
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

7. 用Postman测试API

与第 1 章中的做法相同，成功运行演讲者数据 API 后，我们将使用 Postman 来测试第一个演讲者的数据。在 Postman 的 GUI 中执行以下操作。

- 输入 URL：http://localhost:3000/speakers/1。
- 在 HTTP 方法中选择 GET。
- 点击 Send 按钮。

点击按钮后，可以看到 Postman 中的 GET 请求成功运行，并获取到了 HTTP 200 (OK) 的状

态码，同时 HTTP 响应体文本框中显示了演讲者的 JSON 数据，如图 3-1 所示。

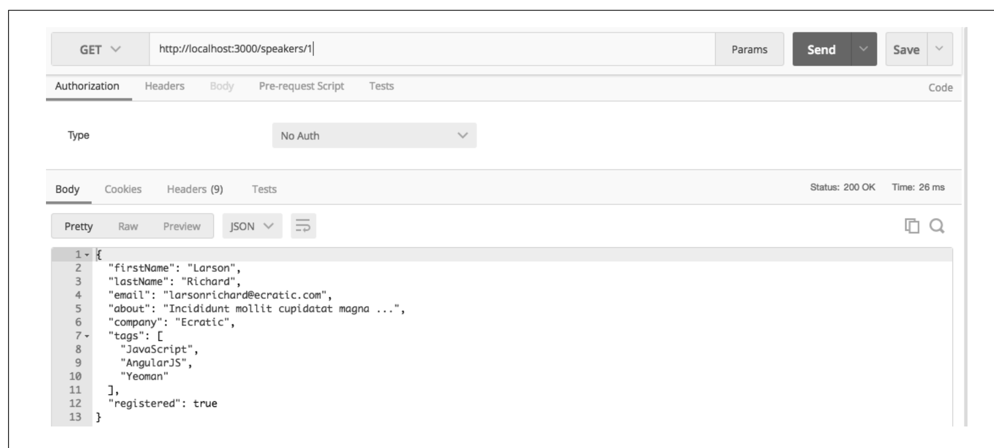


图 3-1：在 Postman 中获取演讲者 JSON 数据

可以在命令行中敲击 Ctrl-C 来关闭 speakers-api-1 应用程序。

3.5.3 speakers-api-2——创建API以提供自定义风格的JSON

除了将 JSON 键名转为驼峰式命名格式，AMS 还提供了其他的 JSON 定制功能。speakers-api-2 这一应用程序将展示如何在 AMS 中定制每个 `speaker` 对象的 JSON 呈现。除了新的 `SpeakerSerializer` 类，speakers-api-2 项目中的所有代码都与之前的 speakers-api-1 项目相同，因此本节仅描述序列化过程。

正式开始前，先安装 gem 来运行 speakers-api-2 项目，如下所示：

```
cd speakers-api-2

bundle exec spring binstub --all
```

1. 用AMS修改JSON呈现

在不修改 `speaker` 对象的情况下，新的 `SpeakerSerializer` 类合并 `first_name` 字段和 `last_name` 字段，从而在 JSON 中产生一个新的 `name` 字段，如例 3-17 所示。

例 3-17 speakers-api-2/app/serializers/speaker_serializer.rb

```
class SpeakerSerializer < ActiveModel::Serializer
  attributes :name, :email, :about,
             :company, :tags, :registered

  def name
    "#{object.first_name} #{object.last_name}"
  end
end
```

对于这一示例有以下两点需要注意。

- `attributes` 定义了 `name` 字段，去除了 `first_name` 和 `last_name`。
- 在 `name` 方法中：
  - `object` 变量表示正在进行 JSON 序列化的 `speaker` 对象；
  - 使用了字符串拼接操作将 `first_name` 和 `last_name` 字段合并为 `name` 字段，原始的 `Speaker` 模型并不会意识到该 `name` 属性的存在。

通过 `attributes` 对 JSON 呈现进行自定义操作是非常强大的，因为它成功地在模型和 JSON 输出之间进行了解耦。

2. 部署API

在 speakers-api-2 目录中运行 `rails s` 命令，将 API 部署到 http://localhost:3000/speakers 这一 URL 上。

3. 用Postman测试API

在 Postman 的 GUI 中向 http://localhost:3000/speakers/1 发起 HTTP `GET` 请求，得到的结果如图 3-2 所示。

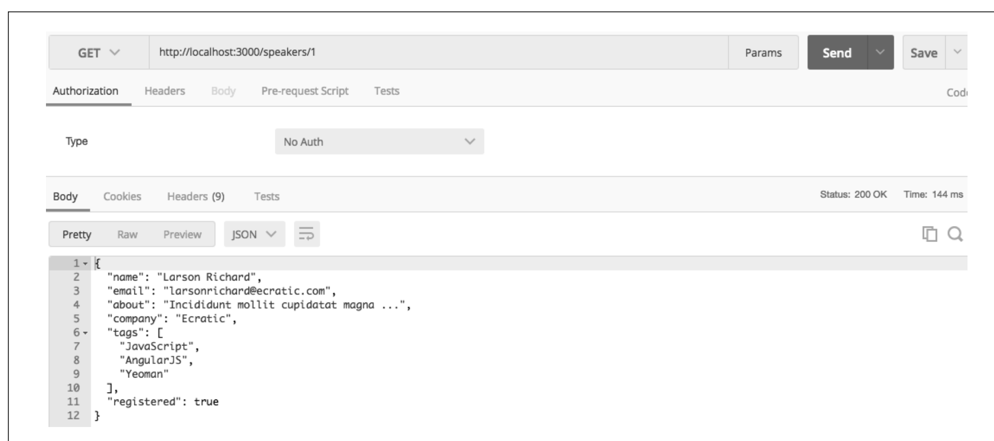


图 3-2：在 Postman 中获取自定义的演讲者 JSON 数据

继续阅读前，在命令行中敲击 Ctrl-C 来关闭 speakers-api-2 应用程序。

3.5.4　有关Rails和Rails API的更多学习资料

本章只介绍了建立简单的 API 应用程序所需要的 Rails 操作和 AMS 知识。如需继续深入学习，可参考以下学习资料：

- Michael Hartl 所著的《Ruby on Rails 教程》[3]；
- Daniel Kehoe 所著的 *Learn Ruby on Rails 5*；

注 3：此书第 4 版已由人民邮电出版社出版。——编者注

- Abraham Kuri 所著的 *APIs on Rails: Building REST APIs with Rails*；
- Chris Kottam 撰写的文章“Get Up and Running with Rails API”；
- Kendra Uzia 撰写的文章“Active Model Serializers, Rails, and JSON! OH MY!”。

3.6 本章回顾

本章先描述了 Ruby 和 JSON 间的简单转换工作，讨论了 JSON 中驼峰式命名格式的重要性，然后演示了如何对一个基于 JSON 的模拟 Web API 进行调用（并使用 Minitest 来测试调用结果）。最后使用 Rails 5 创建了一个 RESTful API，并用 Postman 对其进行了测试。

3.7 内容预告

学习了如何使用 Ruby on Rails 开发基于 JSON 的应用程序后，第 4 章将介绍 JSON 在 Java 和 Spring Boot 中的使用情况。

第 4 章

在Java中使用JSON

前面的章节已经介绍了 JSON 在 JavaScript 和 Ruby on Rails 中的使用，本章将展示 JSON 在 Java（第三个也是最后一个平台）中的使用情况。本章内容包括：

- 在 Java 中用 Jackson 执行 JSON 的序列化 / 反序列化操作；
- 使用 Java 对象和 JSON；
- 在 JUnit 中使用 JSON；
- 发起 RESTful API 调用，并用 JUnit 和 JsonUnit 测试调用结果；
- 用 Spring Boot 搭建简单的 JSON API。

在本章的应用示例中，我们将对第 1 章中部署到 `json-server` 上的数据进行 RESTful API 调用。然后创建一个更真实的基于 JSON 的 Web API。开发 RESTful API 前，我们先来了解一下 Java 中 JSON 序列化 / 反序列化方面的基础知识。

4.1　安装Java和Gradle

本章使用 Gradle 对源代码进行构建和测试。如尚未安装 Java 和 Gradle，请参考 A.5 节和 A.5.2 节中的内容来搭建开发环境，之后即可运行本章中的示例。

4.2　Gradle概览

Gradle 继承了 Apache Ant 和 Maven 等早期 Java 构建系统的理念，并得到了广泛的应用。Gradle 可以为 Java 项目提供以下功能。

- 通用的、标准化的项目目录结构。
- 对 JAR 文件进行依赖管理。

- 统一的构建流程。

`gradle init` 命令可以用于对项目进行初始化：创建核心目录结构、初始化构建脚本、提供一些简单的 Java 源代码和测试代码。以下是 Gradle 项目中的一些关键目录和文件。

- src/main/ 目录包含源代码和资源文件。
  - java/ 目录包含 Java 源代码。
  - resources/ 目录包含源代码所使用的资源文件（属性配置文件、JSON 等数据文件）。
- test/main/ 目录包含测试代码和测试资源文件。
  - java/ 目录包含 Java 测试代码。
  - resources/ 目录包含测试代码所使用的资源文件（属性配置文件、JSON 等数据文件）。
- build/ 目录包含编译源代码和测试代码后生成的 .class 文件。
  - libs/ 目录包含项目构建后生成的 JAR 包和 WAR 包。
- gradlew 是一个 Gradle 封装工具，允许以可执行 JAR 包的方式运行项目。对此，我们将在之后介绍 Spring Boot 时进行详细的描述。
- build.gradle 文件由 `gradle init` 命令生成，但其中的项目依赖则需要手动添加。Gradle 并没有使用 XML，而是使用了一种基于 Groovy 的领域专用语言来编写其构建脚本。
- build/ 目录包含由 `gradle build` 和 `gradle test` 命令所生成的与构建相关的文件。

以下是使用 Gradle 时必须了解的一些 Gradle 任务。可以通过在命令行中执行 `gradle tasks` 来列举这些任务。

`gradle build`
: 构建项目。

`gradle classes`
: 编译 Java 源代码。

`gradle clean`
: 删除 build 目录。

`gradle jar`
: 编译 Java 源代码，并将编译结果和资源文件一同打包为 JAR 包。

`gradle javadoc`
: 根据 Java 源代码生成 JavaDoc 文档。

`gradle test`
: 编译 Java 源代码和测试代码，然后运行单元测试。

`gradle testClasses`
: 编译 Java 测试代码。

以下是示例项目的创建过程。

- 使用 `gradle init --type java-application` 命令初始化 speakers-test 和 speakers-web 应用程序。

- 初始生成的 build.gradle 文件、Java 应用程序和测试文件都只是模拟的，为了运行本章中的示例，它们会被真实代码所替换。

Gradle 的文档非常全面，以下是用于深入了解 Gradle 的一些教程和参考资料：

- Gradle 用户指南；
- Gradle 入门；
- Tim Berglund 所著 *Gradle Beyond the Basics*（O'Reilly 出版社）。

了解了 Gradle 的一些基础知识后，接下来本章会介绍 Java 中的 JSON 类库，然后再看一下相关的代码示例。

4.3　使用JUnit即可完成单元测试

JUnit 是一个广泛使用的单元测试框架。因为在 Java 社区中广为接受，所以本章中的测试都将使用 JUnit 来编写。JUnit 是过程式的，因此使用 JUnit 所编写的单元测试遵循的是 TDD 风格。如果需要在 JUnit 中编写 BDD 风格的测试，那么 Cucumber 会是一个不错的选择。如需学习更多有关 BDD 和 Cucumber 方面的知识，可以参考 Micha Kops 的一篇高质量文章"BDD Testing with Cucumber, Java and JUnit"。

4.4　Java中的JSON类库

Java 中存在多个不错的 JSON 类库，可用于进行 JSON 的序列化 / 反序列化操作，举例如下。

Jackson
　可以在 Jackson 的 GitHub 主页上了解到更多细节信息。

Gson
　Gson 是一个由 Google 提供的 JSON 类库。

JSON-java
　该类库由 Doug Crockford 所提供 。

Java SE（标准版）
　JavaEE 7 以 Java 标准请求（JSR）353 的形式引入了对 JSON 的支持。JSR-353 是一个单独的实现，因此可以将其整合到已有的 Java SE 应用程序中，以作为 Java SE 8 的一部分。作为 Java 增强（JEP）198 号提案的一部分，Java SE 9 将对 JSON 提供原生支持。

出于以下原因，本章中的示例使用的都是 Jackson：

- 在 Java 社区尤其是 Spring 社区中，Jackson 的使用非常广泛；
- 可以提供高质量的功能；
- 经历过时间的考验；
- 开发社区活跃，项目维护较好；
- 文档质量较高。

另外，只使用一种 JSON 类库可以使得我们将注意力集中于 JSON。如上所述，其他类库也都能很好地工作，因此你可以自行选择尝试。

首先，我们来看一下 Java 中的一些基本序列化 / 反序列化操作。

4.5 用Jackson进行JSON序列化/反序列化操作

Java 应用程序需要能够将 Java 数据结构转换为 JSON（序列化），也需要能够进行相反的操作（反序列化）。

4.5.1 对Java中的简单数据类型进行序列化/反序列化操作

与之前的章节一样，我们先对一些基本的 Java 数据类型进行序列化操作：

- 整数型
- 字符串
- 数组
- 布尔值

例 4-1 展示了一个简单的单元测试程序，该程序使用 Jackson 和 JUnit 4 对 Java 中的简单数据类型进行序列化 / 反序列化操作。

例 4-1 speakers-test/src/test/java/org/jsonatwork/ch4/BasicJsonTypesTest.java

```
package org.jsonatwork.ch4;

import static org.junit.Assert.*;

import java.io.*;
import java.util.*;

import org.junit.Test;

import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.core.type.*;
import com.fasterxml.jackson.databind.*;

public class BasicJsonTypesTest {
  private static final String TEST_SPEAKER = "age = 39\n" +
    "fullName = \"Larson Richard\"\n" +
    "tags = [\"JavaScript\",\"AngularJS\",\"Yeoman\"]\n" +
    "registered = true";

  @Test
  public void serializeBasicTypes() {
    try {
      ObjectMapper mapper = new ObjectMapper();
      Writer writer = new StringWriter();
      int age = 39;
      String fullName = new String("Larson Richard");
      List<String> tags = new ArrayList<String>(
```

```
            Arrays.asList("JavaScript", "AngularJS", "Yeoman"));

        boolean registered = true;
        String speaker = null;

        writer.write("age = ");
        mapper.writeValue(writer, age);
        writer.write("\nfullName = ");
        mapper.writeValue(writer, fullName);
        writer.write("\ntags = ");
        mapper.writeValue(writer, tags);
        writer.write("\nregistered = ");
        mapper.configure(SerializationFeature.INDENT_OUTPUT, true);
        mapper.writeValue(writer, registered);
        speaker = writer.toString();
        System.out.println(speaker);
        assertTrue(TEST_SPEAKER.equals(speaker));
        assertTrue(true);
      } catch (JsonGenerationException jge) {
        jge.printStackTrace();
        fail(jge.getMessage());
      } catch (JsonMappingException jme) {
        jme.printStackTrace();
        fail(jme.getMessage());
      } catch (IOException ioe) {
        ioe.printStackTrace();
        fail(ioe.getMessage());
      }
    }

    @Test
    public void deSerializeBasicTypes() {
      try {
        String ageJson = "{ \"age\": 39 }";
        ObjectMapper mapper = new ObjectMapper();
        Map<String, Integer> ageMap = mapper.readValue(ageJson,
           new TypeReference<HashMap<String,Integer>>() {});

        Integer age = ageMap.get("age");

        System.out.println("age = " + age + "\n\n\n");
        assertEquals(39, age.intValue());
        assertTrue(true);
      } catch (JsonMappingException jme) {
        jme.printStackTrace();
        fail(jme.getMessage());
      } catch (IOException ioe) {
        ioe.printStackTrace();
        fail(ioe.getMessage());
      }
    }

  }
```

在以上示例中，由于 `@Test` 注解的声明，JUnit 会将 `serializeBasicTypes()` 和 `deSerializeBasicTypes()` 方法作为测试的一部分来运行。对于 JSON 数据自身来说，这些单元测试用例并未执行太多的断言操作。在介绍对 Web API 的测试工作时，我们会更详细地探讨断言。

以下是 Jackson 中用于 JSON 序列化 / 反序列化操作的一些最重要的类和方法。

- `ObjectMapper` 类负责在 Java 和 JSON 间进行相互转换。
- `ObjectMapper.writeValue()` 方法负责将 Java 数据类型转换为 JSON（在本例中，转换结果被输出到 `Writer` 对象中）。
- `ObjectMapper.readValue()` 方法负责将 JSON 转换为 Java 数据结构。

在命令行中执行以下命令，以运行这一单元测试：

```
cd chapter-4/speakers-test

+gradle test --tests org.jsonatwork.ch4.BasicJsonTypesTest+
```

可以看到如下结果：

```
json-at-work => gradle test --tests org.jsonatwork.ch4.BasicJsonTypesTest
:compileJava
:processResources NO-SOURCE
:classes
:compileTestJava
:processTestResources
:testClasses
:test

org.jsonatwork.ch4.BasicJsonTypesTest > deSerializeBasicTypes STANDARD_OUT
    age = 39



org.jsonatwork.ch4.BasicJsonTypesTest > serializeBasicTypes STANDARD_OUT
    age = 39
    fullName = "Larson Richard"
    tags = ["JavaScript","AngularJS","Yeoman"]
    registered = true

BUILD SUCCESSFUL
```

因为只是序列化 / 反序列化了简单的数据类型，所以以上示例并没有提供什么有意思的功能。但对于 Java 对象来说，序列化 / 反序列化操作就比较有用了。

4.5.2　对Java对象进行序列化/反序列化操作

了解了 Jackson 以及简单数据类型上的 Jackson 操作后，我们将深入介绍 Jackson 对对象的处理。例 4-2 展示了如何使用 Jackson 来序列化 / 反序列化 `speaker` 对象，同时也展示了如何将 JSON 文件反序列化为多个 `speaker` 对象。

例 4-2　speakers-test/src/test/java/org/jsonatwork/ch4/ SpeakerJsonFlatFileTest.java

```
package org.jsonatwork.ch4;

import static org.junit.Assert.*;

import java.io.*;
import java.net.*;
import java.util.*;

import org.junit.Test;

import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import com.fasterxml.jackson.databind.type.*;

public class SpeakerJsonFlatFileTest {

private static final String SPEAKER_JSON_FILE_NAME = "speaker.json";
private static final String SPEAKERS_JSON_FILE_NAME = "speakers.json";
private static final String TEST_SPEAKER_JSON = "{\n" +
    "  \"id\" : 1,\n" +
    "  \"age\" : 39,\n" +
    "  \"fullName\" : \"Larson Richard\",\n" +
    "  \"tags\" : [ \"JavaScript\", \"AngularJS\", \"Yeoman\" ],\n" +
    "  \"registered\" : true\n" +
  "}";

@Test
public void serializeObject() {
    try {
      ObjectMapper mapper = new ObjectMapper();
      Writer writer = new StringWriter();
      String[] tags = {"JavaScript", "AngularJS", "Yeoman"};
      Speaker speaker = new Speaker(1, 39, "Larson Richard", tags, true);
      String speakerStr = null;

      mapper.configure(SerializationFeature.INDENT_OUTPUT, true);
      speakerStr = mapper.writeValueAsString(speaker);
      System.out.println(speakerStr);
      assertTrue(TEST_SPEAKER_JSON.equals(speakerStr));
      assertTrue(true);
    } catch (JsonGenerationException jge) {
      jge.printStackTrace();
      fail(jge.getMessage());
    } catch (JsonMappingException jme) {
      jme.printStackTrace();
      fail(jme.getMessage());
    } catch (IOException ioe) {
      ioe.printStackTrace();
      fail(ioe.getMessage());
    }
  }

  private File getSpeakerFile(String speakerFileName) throws URISyntaxException {
```

```
  ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
  URL fileUrl = classLoader.getResource(speakerFileName);
  URI fileUri = new URI(fileUrl.toString());
  File speakerFile = new File(fileUri);

  return speakerFile;
}

@Test
public void deSerializeObject() {
  try {
    ObjectMapper mapper = new ObjectMapper();
    File speakerFile = getSpeakerFile(
        SpeakerJsonFlatFileTest.SPEAKER_JSON_FILE_NAME);

    Speaker speaker = mapper.readValue(speakerFile, Speaker.class);

    System.out.println("\n" + speaker + "\n");
    assertEquals("Larson Richard", speaker.getFullName());
    assertEquals(39, speaker.getAge());
    assertTrue(true);
  } catch (URISyntaxException use) {
    use.printStackTrace();
    fail(use.getMessage());
  } catch (JsonParseException jpe) {
    jpe.printStackTrace();
    fail(jpe.getMessage());
  } catch (JsonMappingException jme) {
    jme.printStackTrace();
    fail(jme.getMessage());
  } catch (IOException ioe) {
    ioe.printStackTrace();
    fail(ioe.getMessage());
  }
}

@Test
public void deSerializeMultipleObjects() {
  try {
    ObjectMapper mapper = new ObjectMapper();
    File speakersFile = getSpeakerFile(
        SpeakerJsonFlatFileTest.SPEAKERS_JSON_FILE_NAME);

    JsonNode arrNode = mapper.readTree(speakersFile).get("speakers");
    List<Speaker> speakers = new ArrayList<Speaker>();
      if (arrNode.isArray()) {
        for (JsonNode objNode : arrNode) {
        System.out.println(objNode);
        speakers.add(mapper.convertValue(objNode, Speaker.class));
      }
    }

    assertEquals(3, speakers.size());
    System.out.println("\n\n\nAll Speakers\n");
    for (Speaker speaker: speakers) {
```

```
          System.out.println(speaker);
        }

        System.out.println("\n");
        Speaker speaker3 = speakers.get(2);
        assertEquals("Christensen Fisher", speaker3.getFullName());
        assertEquals(45, speaker3.getAge());
        assertTrue(true);
      } catch (URISyntaxException use) {
        use.printStackTrace();
        fail(use.getMessage());
      } catch (JsonParseException jpe) {
        jpe.printStackTrace();
        fail(jpe.getMessage());
      } catch (JsonMappingException jme) {
        jme.printStackTrace();
        fail(jme.getMessage());
      } catch (IOException ioe) {
        ioe.printStackTrace();
        fail(ioe.getMessage());
      }
    }

  }
```

对于以上的 JUnit 单元测试，以下几点值得一提。

- serializeObject() 方法创建了一个 Speaker 对象，然后使用 ObjectMapper.writeValueAsString() 和 System.out.println() 将其序列化，并打印到标准输出。测试代码将 SerializationFeature.INDENT_OUTPUT 设置为 true，以优化 JSON 输出中的缩进和显示。
- deSerializeObject() 方法调用 getSpeakerFile() 来读取包含 speaker 对象的 JSON 输入文件，然后使用 ObjectMapper.readValue() 将其反序列化为 SpeakerJava 对象。
- deSerializeMultipleObjects() 方法执行了以下操作。
  - 调用 getSpeakerFile() 来读取包含 speaker 对象数组的 JSON 输入文件。
  - 调用 ObjectMapper.readTree() 来获取 JsonNode 对象，该对象指向文件中的 JSON 文档的根节点。
  - 访问 JSON 树中的每个节点，并使用 ObjectMapper.convertValue() 方法将节点中的 speaker 数据反序列化为 Java 中的 Speaker 对象。
  - 打印列表中的所有 Speaker 对象。
- getSpeakerFile() 方法会查找类路径中相应的文件，并执行以下操作。
  - 从当前执行线程中获取 ContextClassLoader 对象。
  - 使用 ClassLoader.getResource() 方法从当前类路径中查找相关文件资源。
  - 根据文件名的 URI 创建 File 对象。

之前的测试用例均使用了 JUnit 中的断言方法来测试 JSON 序列化 / 反序列化的结果。

在命令行中执行 gradle test --tests org.jsonatwork.ch4.SpeakerJsonFlatFileTest 命令，运行测试后可以看到以下结果：

```
json-at-work => gradle test --tests org.jsonatwork.ch4.SpeakerJsonFlatFileTest
:compileJava UP-TO-DATE
:processResources NO-SOURCE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test

org.jsonatwork.ch4.SpeakerJsonFlatFileTest > serializeObject STANDARD_OUT
    {
      "id" : 1,
      "age" : 39,
      "fullName" : "Larson Richard",
      "tags" : [ "JavaScript", "AngularJS", "Yeoman" ],
      "registered" : true
    }

org.jsonatwork.ch4.SpeakerJsonFlatFileTest > deSerializeObject STANDARD_OUT

    Speaker [id=1, age=39, fullName=Larson Richard, tags=[JavaScript, AngularJS, Yeoman], registered=true]


org.jsonatwork.ch4.SpeakerJsonFlatFileTest > deSerializeMultipleObjects STANDARD_OUT
    {"id":1,"fullName":"Larson Richard","tags":["JavaScript","AngularJS","Yeoman"],"age":39,"registered":true}
    {"id":2,"fullName":"Ester Clements","tags":["REST","Ruby on Rails","APIs"],"age":29,"registered":true}
    {"id":3,"fullName":"Christensen Fisher","tags":["Java","Spring","Maven","REST"],"age":45,"registered":false}



    All Speakers

    Speaker [id=1, age=39, fullName=Larson Richard, tags=[JavaScript, AngularJS, Yeoman], registered=true]
    Speaker [id=2, age=29, fullName=Ester Clements, tags=[REST, Ruby on Rails, APIs], registered=true]
    Speaker [id=3, age=45, fullName=Christensen Fisher, tags=[Java, Spring, Maven, REST], registered=false]



BUILD SUCCESSFUL
```

除了本章中所介绍的内容，Jackson 还提供了很多其他功能。如需了解更多教程，可参考以下资源：

- Eugen Paraschiv 撰写的“Java Jackson Tutorial”；
- Tutorials Point 网站提供的“Jackson Tutorial”；
- Pankaj 在 JournalDev 网站上撰写的“Jackson JSON Java Parser API Example Tutorial”；
- Mithil Shah 撰写的“Java JSON Jackson Introduction”。

4.6　用模拟API进行单元测试

到目前为止，本章中的 JUnit 测试代码所测试的都是 JSON 静态文件中的数据。接下来我们将针对 API 编写更为实际的测试程序。不过，对于待测试的 API，我们不希望编写太多代码，也不希望在基础架构上耗费大量精力。本节将展示在不编写任何代码的情况下，如何创建一个能提供 JSON 响应的简单的模拟 API。

4.6.1　测试数据

我们将使用之前章节中的演讲者数据作为测试数据（可在 GitHub 上找到），并将其部署为 RESTful API，以完成模拟 API 的创建工作。我们会将 Node.js 模块 `json-server` 用作服务器，向外以 Web API 的形式暴露 speakers.json 文件中的数据。如需了解 `json-server` 的安

装，可参考 A.2.5 节中的内容。以下是在本地机器中新开命令行窗口来使用 5000 端口运行 `json-server` 的步骤：

```
cd chapter-4/speakers-test/src/test/resources

json-server -p 5000 ./speakers.json
```

访问该 API 时，还可以通过在 URI（如 http://localhost:5000/speakers/1）中添加 `id` 来获取单个演讲者的数据。准备好模拟 API 后就可以开始编写单元测试了。

4.6.2　用JUnit对API提供的JSON进行测试

本节的单元测试将实现以下功能：

- 向模拟的演讲者数据 API 发起 HTTP 调用；
- 检查 HTTP 响应体中的值，并将其与预期结果进行对比。

与之前的章节一样，本节会继续使用 Unirest 这一 API 封装工具，但在选择具体类库时会使用 Unirest 的 Java 实现。

在本章前面的 JUnit 单元测试中，测试代码仅检查了最基本的功能（确保没有异常抛出），本节将编写更为复杂的单元测试用例。以下单元测试会检查 HTTP 响应中返回的 JSON 内容，校验其是否与预期结果一致。对于这一检查校验工作，我们可以自行编写代码对数据进行遍历比较，也可以使用第三方类库来减少工作量。JsonUnit 这一类库中就包含了很多有用的匹配规则，可以有效简化 JUnit 单元测试中的 JSON 比较工作。在本节的单元测试中，我们会介绍 JsonUnit 的一些基本用途，但除了本节示例中所涉及的内容外，JsonUnit 还可以提供更多功能，具体包括：

- 正则表达式；
- 更多的匹配规则；
- 可以忽略特定的字段。

例 4-3 中的单元测试通过调用模拟 API 整合了我们之前所介绍的所有内容，并比较了 JSON 响应与预期结果。

例 4-3　speakers-test/src/test/java/org/jsonatwork/ch4/SpeakersJsonApiTest.java

```java
package org.jsonatwork.ch4;

import static org.junit.Assert.*;

import java.io.*;
import java.net.*;
import java.util.*;

import org.apache.http.*;
import org.junit.Test;

import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import com.mashape.unirest.http.HttpResponse;
import com.mashape.unirest.http.Unirest;
```

```
import com.mashape.unirest.http.exceptions.*;
import com.mashape.unirest.request.*;

import static net.javacrumbs.jsonunit.fluent.JsonFluentAssert.assertThatJson;

public class SpeakersApiJsonTest {
  private static final String SPEAKERS_ALL_URI = "http://localhost:5000/speakers";
  private static final String SPEAKER_3_URI = SPEAKERS_ALL_URI + "/3";

  @Test
  public void testApiAllSpeakersJson() {
    try {
      String json = null;
      HttpResponse <String> resp = Unirest.get(
        SpeakersApiJsonTest.SPEAKERS_ALL_URI).asString();

      assertEquals(HttpStatus.SC_OK, resp.getStatus());
      json = resp.getBody();
      System.out.println(json);
      assertThatJson(json).node("").isArray();
      assertThatJson(json).node("").isArray().ofLength(3);
      assertThatJson(json).node("[0]").isObject();
      assertThatJson(json).node("[0].fullName")
              .isStringEqualTo("Larson Richard");
      assertThatJson(json).node("[0].tags").isArray();
      assertThatJson(json).node("[0].tags").isArray().ofLength(3);
      assertThatJson(json).node("[0].tags[1]").isStringEqualTo("AngularJS");
      assertThatJson(json).node("[0].registered").isEqualTo(true);
      assertTrue(true);
    } catch (UnirestException ue) {
      ue.printStackTrace();
    }
  }

  @Test
  public void testApiSpeaker3Json() {
    try {
      String json = null;
      HttpResponse <String> resp = Unirest.get(
        SpeakersApiJsonTest.SPEAKER_3_URI).asString();

      assertEquals(HttpStatus.SC_OK, resp.getStatus());
      json = resp.getBody();
      System.out.println(json);
      assertThatJson(json).node("").isObject();
      assertThatJson(json).node("fullName")
              .isStringEqualTo("Christensen Fisher");
      assertThatJson(json).node("tags").isArray();
      assertThatJson(json).node("tags").isArray().ofLength(4);
      assertThatJson(json).node("tags[2]").isStringEqualTo("Maven");
      assertTrue(true);
    } catch (UnirestException ue) {
      ue.printStackTrace();
    }
  }

}
```

对于以上单元测试，需要注意以下几点。

- `testApiAllSpeakersJson()` 测试用例：
  - 使用 `Unirest.get()` 方法调用 http://localhost:5000/speakers，从演讲者数据 API 中获取所有演讲者的列表；
  - 验证 HTTP 响应状态码为 `OK` (200)；
  - 从 HTTP 响应体中获取包含 `speaker` 对象数组的 JSON 文档；
  - 使用 JSONUnit 的 `assertThatJson()` 方法编写一系列断言语句对 JSON 文档进行校验，具体的校验内容如下：
    - ♦ JSON 文档表示包含 3 个 `speaker` 对象的数组；
    - ♦ 每个 `speaker` 对象中的所有字段值（如 `fullName`，`tags` 和 `registered`）都符合预期值。
  - 运行 `gradle test` 命令后可以看到以下结果。

```
org.jsonatwork.ch4.SpeakersApiJsonTest > testApiAllSpeakersJson STANDARD_OUT
    [
      {
        "id": 1,
        "fullName": "Larson Richard",
        "tags": [
          "JavaScript",
          "AngularJS",
          "Yeoman"
        ],
        "age": 39,
        "registered": true
      },
      {
        "id": 2,
        "fullName": "Ester Clements",
        "tags": [
          "REST",
          "Ruby on Rails",
          "APIs"
        ],
        "age": 29,
        "registered": true
      },
      {
        "id": 3,
        "fullName": "Christensen Fisher",
        "tags": [
          "Java",
          "Spring",
          "Maven",
          "REST"
        ],
        "age": 45,
        "registered": false
      }
    ]

BUILD SUCCESSFUL
```

- `testApiSpeaker3Json()` 测试用例：
  - 使用 `Unirest.get()` 方法调用 http://localhost:5000/speakers/3，从演讲者数据 API 中获取第 3 个演讲者的数据；
  - 校验 HTTP 响应状态码为 `OK` (200)；

– 从 HTTP 响应体中获取包含单个 speaker 对象的 JSON 文档；
– 使用 JSONUnit 的 assertThatJson() 方法编写一系列断言语句对 JSON 文档进行校验，具体的校验内容如下：
    ♦ JSON 文档表示单个的 speaker 对象；
    ♦ speaker 对象中的所有字段值都符合预期；
– 运行 gradle test 命令后可以看到以下结果。

```
org.jsonatwork.ch4.SpeakersApiJsonTest > testApiSpeaker3Json STANDARD_OUT
    {
      "id": 3,
      "fullName": "Christensen Fisher",
      "tags": [
        "Java",
        "Spring",
        "Maven",
        "REST"
      ],
      "age": 45,
      "registered": false
    }
```

以上单元测试只使用了 Unirest 中 Java 类库的基本功能。除此之外，Unirest 的 Java 类库还提供了以下特性：

- 支持所有的 HTTP 方法（GET、POST、PUT、DELETE、PATCH）；
- 将 HTTP 响应体转换为 Java 对象的过程中可以自定义转换规则；
- 发送异步（即非阻塞）请求；
- 超时；
- 文件上传；
- 更多功能。

如需了解更多信息，可参考 Unirest Java 类库的官方网站。

继续阅读前，可在命令行中敲击 Ctrl-C 来关闭 json-server。

本节展示了如何部署和调用模拟 API，接下来我们将搭建一个小型的 RESTful API。

4.7 用Spring Boot搭建小型Web API

本节将继续使用演讲者数据，通过 Spring Boot 将其创建为 API（示例代码中的 chapter-4/speakers-api）。Spring 框架可以简化 Java Web 应用程序和 RESTful API 的开发、部署工作。Spring Boot 则通过默认配置简化 Spring 应用程序的创建工作。使用 Spring Boot 可以带来以下好处。

- 不再需要繁琐易错的 XML 配置文件。
- Tomcat/Jetty 能够以程序嵌入的方式运行，从而避免单独部署 WAR（Web application ARchive，Web 应用程序存档）包。你仍然**可以**使用 Spring Boot 和 Gradle 来构建 WAR 包，并将其部署到 Tomcat 上。但如本章后面内容所展示的那样，以可执行 JAR 包的方式运行 Web 应用程序可以减少项目在搭建、安装方面的工作，从而简化开发环境，促进应用程序迭代式开发。

我们将通过以下步骤使用 Spring Boot 来创建并部署演讲者数据 API。

(1) 编写源代码。

- 模型
- 控制器
- 应用程序

(2) 创建一个构建脚本（build.gradle）。
(3) 使用 `gradlew` 部署嵌入运行的 JAR 包。
(4) 用 Postman 进行测试。

4.7.1 创建模型

例 4-4 中的 `Speaker` 类是一个普通的 Java 对象，用于表示演讲者数据，API 则会将这些数据以 JSON 的形式加以呈现。

例 4-4 speakers-api/src/main/java/org/jsonatwork/ch4/Speaker.java

```
package org.jsonatwork.ch4;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Speaker {
  private int id;
  private int age;
  private String fullName;
  private List<String> tags = new ArrayList<String>();
  private boolean registered;

  public Speaker() {
    super();
  }

  public Speaker(int id, int age, String fullName, List<String> tags,
                 boolean registered) {
    super();
    this.id = id;
    this.age = age;
    this.fullName = fullName;
    this.tags = tags;
    this.registered = registered;
  }

  public Speaker(int id, int age, String fullName, String[] tags,
                 boolean registered) {
    this(id, age, fullName, Arrays.asList(tags), registered);
  }

  public int getId() {
    return id;
  }
```

```java
    public void setId(int id) {
      this.id = id;
    }

    public int getAge() {
      return age;
    }

    public void setAge(int age) {
      this.age = age;
    }

    public String getFullName() {
      return fullName;
    }

    public void setFullName(String fullName) {
      this.fullName = fullName;
    }

    public List<String> getTags() {
      return tags;
    }

    public void setTags(List<String> tags) {
      this.tags = tags;
    }

    public boolean isRegistered() {
      return registered;
    }

    public void setRegistered(boolean registered) {
      this.registered = registered;
    }

    @Override
    public String toString() {
      return String.format(
        "Speaker [id=%s, age=%s, fullName=%s, tags=%s, registered=%s]",
        id, age, fullName, tags, registered);
    }

  }
```

除了为 `speaker` 对象实例提供数据成员、构造器和 getter/setter 访问方法外，以上代码没有承担太多工作。如后面的程序所示，Spring 会将该 `speaker` 对象自动转换成 JSON，因此上述代码中也不会包含 JSON 格式方面的任何信息。

4.7.2　创建控制器

在 Spring 应用程序中，控制器负责接受 HTTP 请求并返回 HTTP 响应。在本节示例中，演讲者数据的 JSON 会以 HTTP 响应体的形式返回。例 4-5 展示了 `SpeakerController` 类的源代码。

例 4-5　speakers-api/src/main/java/org/jsonatwork/ch4/SpeakerController.java

```java
package org.jsonatwork.ch4;

import java.util.*;
import org.springframework.web.bind.annotation.*;
import org.springframework.http.*;

@RestController
public class SpeakerController {

  private static Speaker speakers[] = {
    new Speaker(1, 39, "Larson Richard",
                new String[] {"JavaScript", "AngularJS", "Yeoman"}, true),
    new Speaker(2, 29, "Ester Clements",
                new String[] {"REST", "Ruby on Rails", "APIs"}, true),
    new Speaker(3, 45, "Christensen Fisher",
                new String[] {"Java", "Spring", "Maven", "REST"}, false)
  };

  @RequestMapping(value = "/speakers", method = RequestMethod.GET)
  public List<Speaker> getAllSpeakers() {
    return Arrays.asList(speakers);
  }

  @RequestMapping(value = "/speakers/{id}", method = RequestMethod.GET)
  public ResponseEntity<?> getSpeakerById(@PathVariable long id) {
    int tempId = ((new Long(id)).intValue() - 1);
    if (tempId >= 0 && tempId < speakers.length) {
      return new ResponseEntity<Speaker>(speakers[tempId], HttpStatus.OK);
    } else {
      return new ResponseEntity(HttpStatus.NOT_FOUND);
    }
  }
}
```

关于这段代码，需要注意以下几点。

- `@RestController` 注解将 `SpeakerController` 类标记为处理 HTTP 请求的 Spring MVC 控制器。
- `speakers` 数组是硬编码的，仅适用于测试。真实的应用程序中会有一个单独的数据层负责从数据库中读取 `speakers` 信息，或者通过外部 API 调用获取 `speakers` 数据。
- `getAllSpeakers()` 方法执行了以下操作。
  - 响应 /speakers URI 上的 HTTP `GET` 请求。
  - 以 `ArrayList` 的形式读取整个 `speakers` 数组，并将其转换成 JSON 数组的格式，然后以 HTTP 响应体的形式输出。
  - `@RequestMapping` 注解会由 `getAllSpeakers()` 方法来处理 /speakers URI 上的 `GET` 请求。
- `getSpeakerById()` 方法执行了以下操作。
  - 响应 /speakers/{id} URI（`id` 表示演讲者的 ID）上的 HTTP `GET` 请求。
  - 根据演讲者的 ID 读取 `speaker` 对象数据，并将其转换成 JSON 对象的格式，然后以 HTTP 响应体的形式输出。

– `@PathVariable` 注解将 HTTP 请求路径中的演讲者 ID 用作 `id` 参数，以查找相应的 `speaker` 对象。
– 返回的 `ResponseEntity` 类型可用于设定 HTTP 响应状态码以及响应中的演讲者数据。

在上述两个方法中，`Speaker` 对象都能自动转换成 JSON，无须任何额外工作。默认情况下，Spring 会使用 Jackson 来自动完成从 Java 到 JSON 的转换操作。

4.7.3 注册应用程序

本章前面提到过，可以将演讲者数据 API 打包成 WAR 包，并将其部署到 Tomcat 等应用程序服务器上。不过，以独立应用程序的方式从命令行中启动 API 会更简单。具体操作如下。

- 添加 Java 中的 `main` 方法。
- 将应用程序打包为可执行 JAR 包。

例 4-6 中的 `Application` 类提供了我们所需要的 `main()` 方法。

例 4-6　speakers-api/src/main/java/org/jsonatwork/ch4/Application.java

```
package org.jsonatwork.ch4;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

在以上示例中，`@SpringBootApplication` 注解在 Spring 上对应用程序进行了注册，并完成了 `SpeakerController` 类与 `Speaker` 类的设定。

以上就是所有的源代码。接下来我们会介绍用于构建应用程序的 build.gradle 脚本。

4.7.4 编写构建脚本

Gradle 使用名为 build.gradle 的脚本文件来构建应用程序。例 4-7 展示了 speakers-api 项目中的这一构建脚本。

例 4-7　speakers-api/build.gradle

```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:1.5.2.RELEASE")
  }
}

apply plugin: 'java'
```

```
apply plugin: 'org.springframework.boot'

ext {
  jdkVersion = "1.8"
}

sourceCompatibility = jdkVersion
targetCompatibility = jdkVersion

tasks.withType(JavaCompile) {
  options.encoding = 'UTF-8'
}

jar {
  baseName = 'speakers-api'
  version =  '0.0.1'
}

repositories {
  mavenCentral()
}

test {
  testLogging {
    showStandardStreams = true // 显示标准输出和标准错误
  }
  ignoreFailures = false
}

dependencies {
  compile (
    [group: 'org.springframework.boot', name: 'spring-boot-starter-web']
  )
}
```

关于以上示例中的 build.gradle 脚本，需要注意以下几点。

- Spring Boot 的 Gradle 插件执行了以下操作。
  - 将所有组件构建为单个可执行 JAR 包。
  - 在可执行 JAR 包中从路径 src/main/java 处搜索包含 `main()` 方法的类（在本例中，这个类即为 `Application.java`）来部署 API。
- `jar` 这一代码块定义了 JAR 包文件的文件名。
- 在 Gradle 中声明 `repositories` 部分代码，使得应用程序从 Maven 中心仓库拉取程序依赖包。
- 在 Gradle 中声明 `testLogging` 部分代码，使得应用程序在执行测试时显示标准结果输出和标准错误输出。
- `dependencies` 声明了 speakers-api 项目所依赖的 JAR 包。

以上的构建脚本还是比较简单的。除此之外，Gradle 还提供了很多非常强大的构建功能。如需学习更多有关 Gradle 构建的知识，可参考 Gradle 用户手册中的“Writing Gradle Build Scripts”部分。

我们已经介绍了构建脚本，现在是时候来部署演讲者数据 API 了。

4.7.5　部署API

`gradle init` 命令在创建 speakers-api 项目时会生成 gradlew 脚本。如需学习更多有关创建 Gradle 项目的知识，可参考 Gradle 用户手册中的“Creating New Gradle Builds”部分。

通过执行以下步骤，gradlew 脚本会组织所有的资源并简化部署工作。

- 调用 build.gradle 脚本来构建应用程序，并使用 Spring Boot 插件将结果打包为可执行 JAR 包。
- 在内嵌的Tomcat服务器上以可执行JAR包的形式将演讲者数据API部署到http://localhost:8080/speakers。

在 speakers-api 目录中运行 `./gradlew bootRun` 命令来部署应用程序，显示日志消息后可以看到以下结果：

```
2017-03-31 16:06:08.975  INFO 23433 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomca
/8.5.11
2017-03-31 16:06:09.084  INFO 23433 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebAppli
ationContext
2017-03-31 16:06:09.084  INFO 23433 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initializ
tion completed in 1288 ms
2017-03-31 16:06:09.215  INFO 23433 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet'
o [/]
2017-03-31 16:06:09.220  INFO 23433 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFil
er' to: [/*]
2017-03-31 16:06:09.221  INFO 23433 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilt
r' to: [/*]
2017-03-31 16:06:09.221  INFO 23433 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFi
ter' to: [/*]
2017-03-31 16:06:09.221  INFO 23433 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter
 to: [/*]
2017-03-31 16:06:09.517  INFO 23433 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.sp
ingframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@3d0f8e03: startup date [Fri Mar 31 16:06:07 MDT 2017]; ro
t of context hierarchy
2017-03-31 16:06:09.599  INFO 23433 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/speakers],methods=[GET]}"
nto public java.util.List<org.jsonatwork.ch4.Speaker> org.jsonatwork.ch4.SpeakerController.getAllSpeakers()
2017-03-31 16:06:09.600  INFO 23433 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/speakers/{id}],methods=[GE
]}" onto public org.springframework.http.ResponseEntity<?> org.jsonatwork.ch4.SpeakerController.getSpeakerById(long)
2017-03-31 16:06:09.604  INFO 23433 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.s
ringframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.BasicError
ontroller.error(javax.servlet.http.HttpServletRequest)
2017-03-31 16:06:09.605  INFO 23433 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]
}" onto public org.springframework.web.servlet.ModelAndView org.springframework.boot.autoconfigure.web.BasicErrorController.errorHtml(java
.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
2017-03-31 16:06:09.643  INFO 23433 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto ha
dler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-03-31 16:06:09.644  INFO 23433 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of
type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-03-31 16:06:09.690  INFO 23433 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] ont
 handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-03-31 16:06:09.863  INFO 23433 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on
startup
2017-03-31 16:06:09.934  INFO 23433 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http
2017-03-31 16:06:09.940  INFO 23433 --- [           main] org.jsonatwork.ch4.Application           : Started Application in 2.536 seconds
JVM running for 2.922)
> Building 80% > :bootRun
```

4.7.6　用Postman测试API

与第 1 章中的做法相同，成功运行演讲者数据 API 后，我们将使用 Postman 对第一个演讲者的数据进行测试。在 Postman 的 GUI 中执行以下操作。

(1) 输入 URL：http://localhost:8080/speakers/1。
(2) 在 HTTP 方法中选择 `GET`。
(3) 点击 Send 按钮。

点击按钮后，可以看到 Postman 中的 `GET` 请求成功运行，并获取了 HTTP 200 (OK) 的状态码，同时 HTTP 响应体文本框中显示了演讲者的 JSON 数据，如图 4-1 所示。

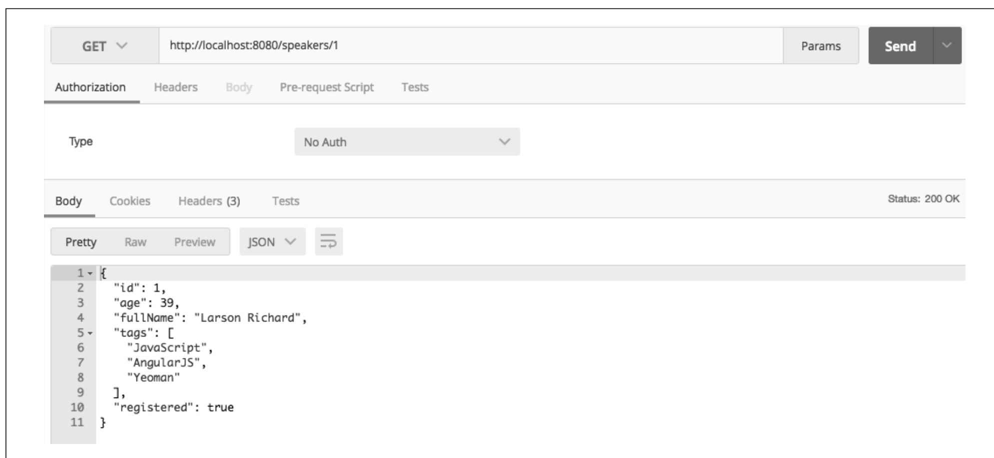


图 4-1：在 Postman 中使用演讲者数据 API

可以在命令行中敲击 Ctrl-C 来关闭 gradlew 程序。

正如之前所描述的那样，由于省略了以下工作，应用程序的开发和部署都得到了简化。

- 创建 / 修改 web.xml 等 Spring 或 Java EE 中的 XML 元数据配置。
- 部署 WAR 文件。
- 安装 Tomcat。

值得一提的是，上述部署步骤仅展示了如何搭建简单的 Web API 开发环境。当在需要共享的环境（测试环境、用户验收环境、生产环境）中进行操作时，还是需要将 WAR 包部署到应用程序服务器中，从而对应用程序进行性能测试和调优。

4.8　本章回顾

本章先描述了 Java 和 JSON 间的简单转换工作，然后演示了如何调用一个基于 JSON 的模拟 Web API，并通过 JUnit 测试来调用结果。最后用 Spring Boot 创建了一个 RESTful API，并用 Postman 对其进行了测试。

4.9　内容预告

我们已经介绍了 JSON 在多个平台（JavaScript、Ruby on Rails 和 Java）中的基本使用情况，接下来的 3 章将深入描述 JSON 生态系统：

- JSON Schema；
- 在 JSON 中进行搜索；
- 对 JSON 进行格式转换。

第 5 章将介绍 JSON Schema 的使用，具体展示如何使用 JSON Schema 对 JSON 文档进行结构化和校验。

第二部分

JSON生态系统

第5章

JSON Schema

前面几章展示了 JSON 在几个核心平台（JavaScript、Ruby on Rails 和 Java）上的基本使用，接下来几章将更深入地介绍 JSON。对于应用程序间用于交换数据的 JSON 文档，本章将展示如何使用 JSON Schema 来定义其结构与格式，具体内容包括：

- JSON Schema 概览；
- JSON Schema 核心——基础知识与工具；
- 如何使用 JSON Schema 来设计和测试 API。

逐步介绍了 JSON Schema 的概念后，我们将在本章的应用示例中使用 JSON Schema 来设计 API。如前言中提到的那样，从第二部分内容开始，所有的示例都将采用 Node.js 来编写，以控制章节篇幅。不过事实上，JSON Schema 也能在其他平台上有效工作。如果尚未安装 Node.js，那么可以参考附录 A 中的相关指导步骤来完成安装操作。

5.1 JSON Schema概览

对于 JSON Schema，很多架构师和开发人员都会感觉比较陌生。因此，在详细描述 JSON Schema 的用法前，有必要先了解一下什么是 JSON Schema，它有什么用，为什么以及何时应当使用它。本节将介绍 JSON Schema 的标准规范，并展示一个简单的示例。

5.1.1 JSON Schema是什么

JSON Schema 是对 JSON 文档 / 消息中的内容、结构与格式的声明。JSON Schema 可以校验 JSON 文档，这可能会导致一些质疑：难道普通的 JSON 校验方法无法完全满足需求？遗憾的是，因为**校验**一词包含了多种含义，所以该问题的答案是肯定的。

5.1.2 语法校验与语义校验

普通的 JSON 校验与 JSON Schema 校验的区别在于校验的**类型**。当使用普通的 JSON 校验时，校验的仅仅是 JSON 文档的语法。这一校验类型只能确保文档的格式正确（即存在对称的大括号、键名由双引号括起来等），因此称为**语法校验**。我们在之前的章节中通过 JSONLint 执行过这一操作，而各个平台的 JSON 解析操作中也包含了该语法校验的内容。

JSON Schema的作用

采用语法校验是一个不错的开始，但有时我们需要通过语义校验对 JSON 进行更深层次的检查。设想以下场景。

- 作为 API 的使用者，你需要检查 API 的 JSON 响应，以确保其包含有效的 `Speaker` 或者 `Orders` 列表。
- 作为 API 的提供者，你需要检查调用请求中的 JSON，以确保其仅包含需要的字段。
- 你需要检查数据格式，例如，检查电话号码、日期 / 时间、邮政编号、电子邮箱、信用卡号码等。

JSON Schema 可以在上述场景中大显身手，相应的校验类型称为**语义校验**。除了检查语法，这种校验方式还会验证数据的**含义**。因为可以帮助定义接口，所以采用 JSON Schema 对 API 的设计来说也是大有裨益的，本章后面将对此进行具体介绍。

5.1.3 简单示例

探讨 JSON Schema 的详细内容前，我们先研究一下例 5-1，以便对 JSON Schema 的语法有个感性认知。

例 5-1 ex-1-basic-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    }
  }
}
```

以上示例中的 Schema 声明了 JSON 文档中包含 3 个字段（`email`、`firstName` 和 `lastName`），每个字段的值均为字符串。本节会忽略该 Schema 的具体语法，但请放心，本章后面会对语法进行详细描述。例 5-2 展示了与上述 Schema 对应的一个 JSON 实例。

例 5-2　ex-1-basic.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard"
}
```

5.1.4　Web上的JSON Schema资源

对于与 JSON Schema 有关的一切资源，图 5-1 所示网站是最好的着手点。该网站包含了大量的文档和示例。

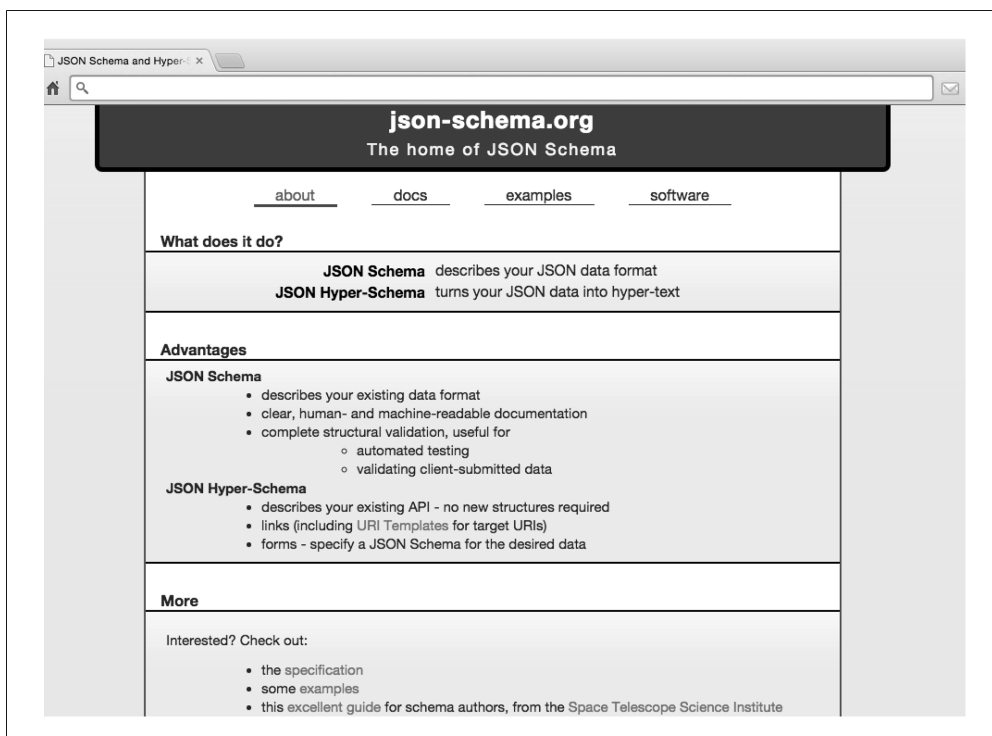


图 5-1：json-schema.org 网站

你可以在该网站上找到 Schema 示例、主流平台上的高质量校验类库，以及维护 JSON Schema 标准的 GitHub 主页，如图 5-2 所示。

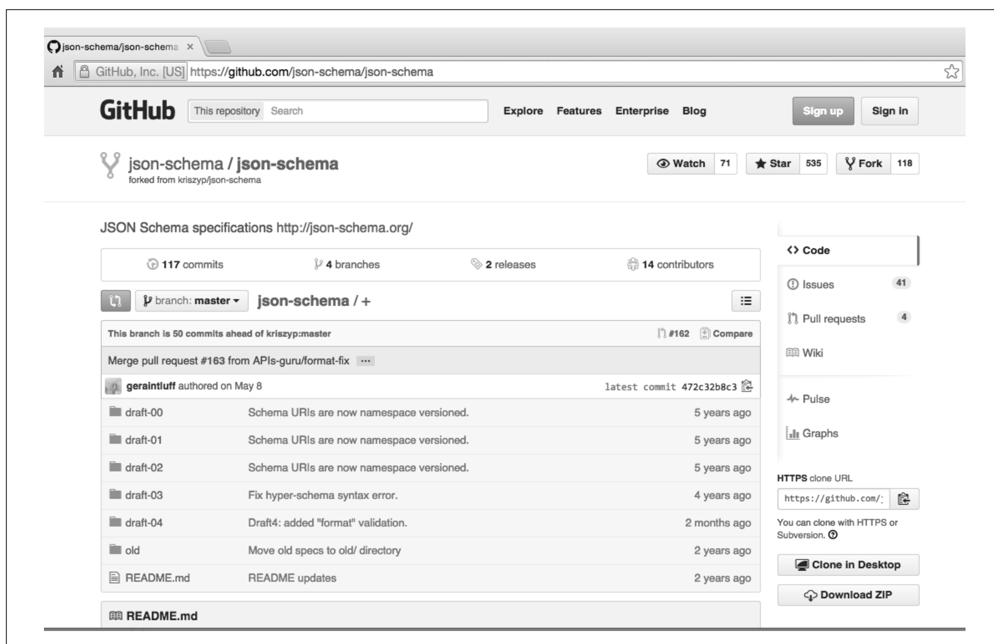


图 5-2：json-schema 的 GitHub 主页

可以在 GitHub 主页上查看 JSON Schema 标准的更新、问题列表以及最新进展（具体详情参见 5.1.7 节）。

5.1.5　为什么使用JSON Schema

JSON Schema 可用于校验 JSON 文档的内容和语义，以下是实际应用中的一些例子。

安全

开放式 Web 应用程序安全项目（Open Web Application Security Project，OWASP）的 Web Service 安全速查表建议：Web Service 应当使用 Schema 来校验服务请求中的数据。诚然，上述文档中谈论的是 XML Schema，但这一建议背后的考虑对 JSON 来说也是适用的。OWASP 号召校验字段长度（最小值 / 最大值）以及字段格式（如电话号码、邮编），以提高服务的安全性。

消息设计

JSON 的应用已经不再局限于 API 领域。在 Apache Kafka 这样的消息系统（第 10 章将对此进行详细介绍）中，很多企业会将 JSON 作为首选的数据交换格式。在消息系统的这种架构风格中，消息的提供者和使用者是彻底解耦的，而 JSON Schema 可以确保使用者收到格式正确的消息。

API 设计

JSON 可以说是 API 设计领域的“一等公民”。通过声明数据文档的格式、内容和结构，JSON Schema 可以帮助定义 API 协议。

原型制作

由于 JSON Schema 的结构化和严谨性，使用 JSON Schema 来制作原型听上去似乎有些违反直觉。在本章后面设计 API 时，我们将展示如何通过 JSON Schema 及相关工具来改进原型制作流程。

5.1.6 我在JSON Schema上的经历

如前言中提到的，2009 年时我并不确定企业是否适合采用 JSON。那时的我喜欢 JSON 的高效与简洁，但无法确保应用程序间交换的 JSON 文档的结构与内容。不过，在 2010 年了解了 JSON Schema 后，我就改变了观点，开始考虑将 JSON 作为企业级数据格式的可行性。

5.1.7 JSON Schema标准的现状

截至本书英文版出版 [1]，JSON Schema 标准规范的版本是**实现草案 4**（v0.4），而下一个**实现草案 6**（v0.6）也正在制订中。草案 5（v0.5）版本于 2016 年年底发布，这一草案只是收集了一些工作进展，因此只是一个**工作草案**，而非**实现草案**。JSON Schema 的版本号是 0.*x*，但完全没必要对此有所顾虑。从本章的示例中可以看到，JSON Schema 十分健壮，可以提供可靠的校验功能，所有的主流编程平台上也都有大量的 JSON Schema 类库可用。如需了解更多细节，可参考 JSON Schema 草案 4 的标准规范。

5.1.8 JSON Schema与XML Schema

除了以下几点，JSON Schema 在 JSON 中的作用与 XML Schema 在 XML 文档中的作用相同：

- JSON 文档不会引用 JSON Schema，根据 Schema 校验 JSON 文档的操作由应用程序负责；
- JSON Schema 不包含命名空间；
- JSON Schema 文件的扩展名为 .json。

5.2 JSON Schema核心——基础知识与工具

我们已经概览过 JSON Schema，现在是时候对其进行更深入的介绍了。JSON Schema 功能强大，但显得繁琐单调，因此本节将介绍一些工具来简化相关工作。之后我们会介绍一些基础的数据类型与核心关键词，以便为实际项目中使用 JSON Schema 夯实基础。

5.2.1 JSON Schema工作流与工具

JSON Schema 的语法可能有些令人怯步，但事实上开发人员可以使用一些优秀的工具来简化相关工作，无须手动编写所有代码。

1. JSON Editor Online

第 1 章已介绍过 JSON Editor Online，但该工具值得我们再次进行简要说明。我们可以使用 JSON Editor Online 对 JSON 文档进行建模，从而对将要构建的数据有个感性的认知；也可

注 1：截至本书中文版出版，JSON Schema v0.7 版本已于 2017 年 11 月 19 日发布。——译者注

以使用这一工具来生成 JSON 文档，从而避免大量的手工输入操作。完成 JSON 文档的生成工作后，可以将 JSON 保存到剪贴板中。

2. JSONSchema.net

确定 JSON 的核心设计后，即可基于前面 JSON Editor Online 所创建的 JSON 文档，使用 JSONSchema.net 应用程序来生成 JSON Schema（如图 5-3 所示）[2]。在创建 Schema 的过程中，单是使用 JSONSchema.net 应用程序即可节省 80% 的手工输入工作。在进行 Schema 相关工作时，我总是以使用 JSONSchema.net 应用程序为开端，之后再逐步修缮。

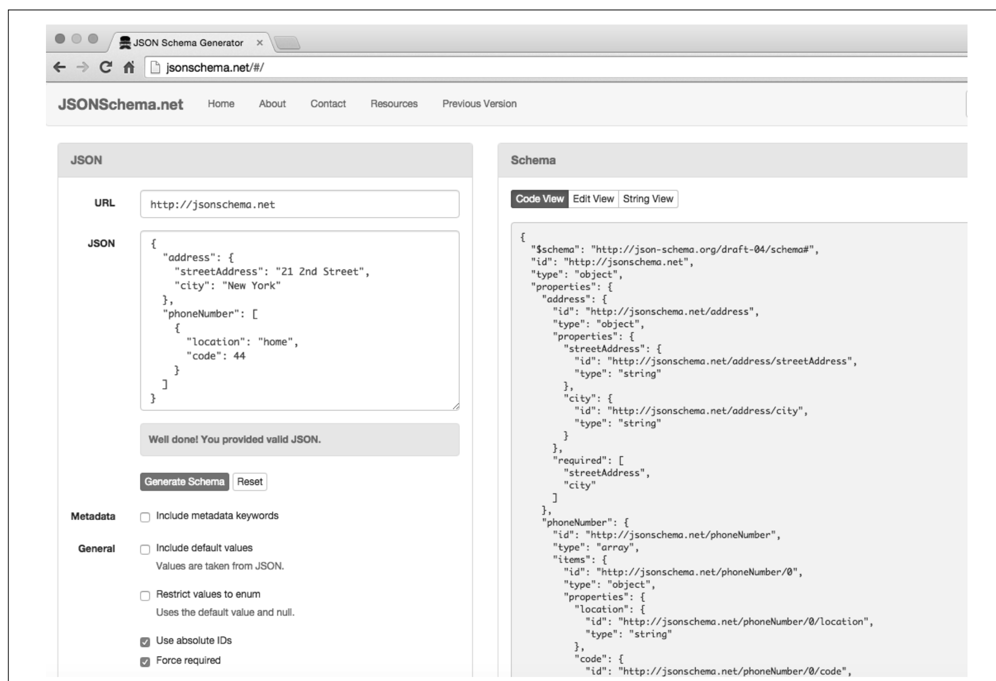


图 5-3：JSONSchema.net 中的演讲者数据 Schema

以下是使用 JSONSchema.net 生成初始 Schema 的步骤。

(1) 在左侧输入框中粘贴 JSON 文档。
(2) 使用默认设置，并进行以下变更。

- 关闭“Use absolute IDs”。
- 关闭“Allow additional properties”。
- 关闭“Allow additional items”。

(3) 点击 Generate Schema 按钮。
(4) 将生成的 Schema 复制到剪贴板中。

注 2：由于 JSONSchema.net 网站已改版，因此相关的截图和操作步骤已不再适用。——译者注

3. JSON Validate

创建 JSON Schema 后，JSON Validate 应用程序可以根据该 Schema 来校验 JSON 文档，如图 5-4 所示。

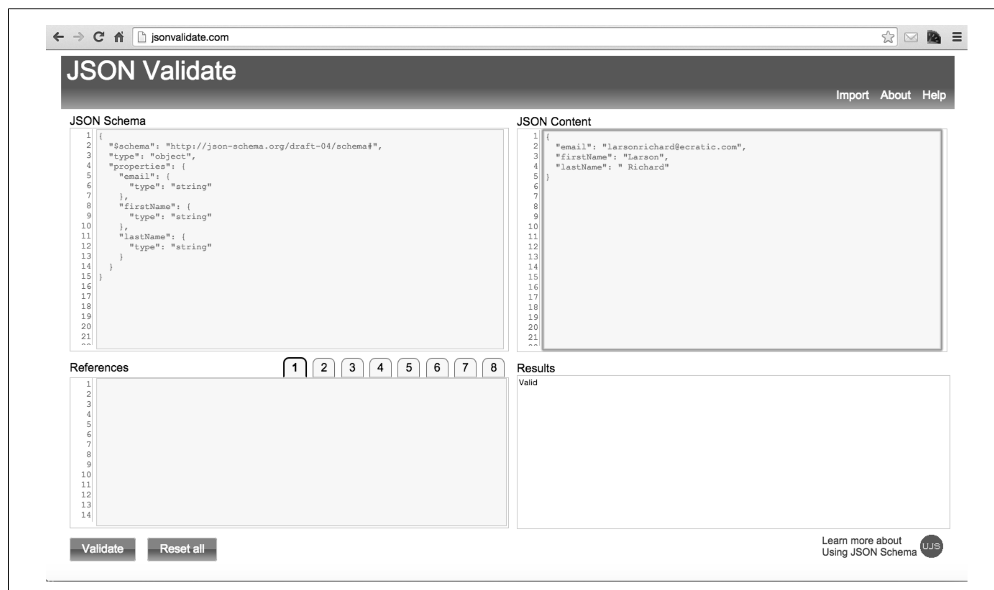


图 5-4：jsonvalidate.com 中合法的演讲者数据 Schema

可以使用 Schema 来校验 JSON 文档，具体步骤如下所示。

(1) 将 JSON 文档和 Schema 粘贴到 JSON Validate 应用程序中的相应地方。
(2) 由于已经不再需要 `id` 字段，因此需从 Schema 中将其全部删除。
(3) 点击 Validate 按钮来校验文档。

4. 在命令行界面使用的NPM模块：`validate`和`jsonlint`

以上在线工具都需要良好的网络连接，但这一点有时很难做到，因此拥有可以在本地运行的工具就显得非常重要。另外，如果数据中包含敏感信息，那么在本机命令行界面中运行会更加安全。`validate` 模块相当于 Node.js 中的 jsonvalidate.com 网站。如需安装并运行 `validate` 模块，可参考 A.2.5 节中的内容。

jsonvalidate.com 网站和 `validate` 模块都是“Using JSON Schema”项目的一部分，该项目提供了非常不错的 Schema 资源，可以在 GitHub 上找到其主页。第 1 章中曾经介绍过 JSONLint 网站，事实上 JSONLint 还可以通过 `jsonlint` 这一 Node.js 模块在命令行中运行。如需安装并运行 `jsonlint` 模块，可参考 A.2.5 节中的内容。

我一般只使用 `jsonlint` 模块的语法校验功能；但如果在命令行中运行 `jsonlint --help`，就可以看到该模块也支持 Schema 语义校验。如需了解更多信息，可参考 `jsonlint` 在 GitHub 上的文档。

我们将在命令行中使用 `validate` 模块来运行接下来的示例。

5.2.2　核心关键词

以下是 JSON Schema 中的核心关键词。

`$schema`

　　声明遵循的 JSON Schema 标准的版本。例如，`"$schema": "http://json-schema.org/draft-04/schema#"` 声明了所属的 schema 遵循 0.4 版本的 JSON Schema 标准，`http://json-schema.org/schema` 则向 JSON 校验器声明所属的 schema 遵循的是当前最新的标准版本（撰写本书时，该版本为 0.4）。因为某些 JSON 校验器默认采用老版本的 JSON Schema，不是最新版的标准，所以使用 `http://json-schema.org/schema` 具有一定的风险。为保险起见，建议在 `$schema` 中始终声明具体的 Schema 版本，以便 JSON 文档和 JSON 校验器确定版本信息。

`type`

　　声明某个字段的数据类型，如 `"type": "string"`。

`properties`

　　声明对象中的字段，其中包含具体字段的 `type` 信息。

5.2.3　基础类型

例 5-3 中的文档包含了前面介绍的 JSON 中的一些基础数据类型（如字符串、数值、布尔值）。

例 5-3　ex-2-basic-types.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "age": 39,
  "postedSlides": true,
  "rating": 4.1
}
```

与第 1 章中的 JSON 核心数据类型（`string`、`number`、`array`、`object`、`boolean`、`null`）一样，JSON Schema 使用了相同的基础数据类型，但添加了 `integer` 类型来表示整数。`number` 类型含义不变，仍然表示整数以及浮点数。

例 5-4 中的 JSON Schema 描述了例 5-3 中的 JSON 文档结构。

例 5-4　ex-2-basic-types-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string"
    },
    "firstName": {
      "type": "string"
    },
```

```
        "lastName": {
          "type": "string"
        },
        "age": {
          "type": "integer"
        },
        "postedSlides": {
          "type": "boolean"
        },
        "rating": {
          "type": "number"
        }
      }
    }
```

在以上示例中，需要注意以下几点。

- `$schema` 字段声明该 schema 遵循的是 JSON Schema v0.4，因此校验 JSON 文档时，应该使用 v0.4 版本的规则。
- 第一个 `type` 字段声明 JSON 文档的根节点为对象，该对象中包含了所有的文档字段。
- `email`、`firstName`、`lastName` 字段的类型是 `string`。
- `age` 的类型为 `integer`。虽然 JSON 本身只支持 `number` 类型，但 JSON Schema 在此基础上增加了更细粒度的 `integer` 类型。`postedSlides` 的类型为 `boolean`。`rating` 的类型为 `number`，因此可以接受浮点数。

使用 `validate` 运行以上示例，可以看到对于例 5-4 中的 Schema 来说，JSON 文档是合法的。

```
json-at-work => validate ex-2-basic-types.json ex-2-basic-types-schema.json
JSON content in file ex-2-basic-types.json is valid
json-at-work =>
```

虽然上述 Schema 的使用是一个不错的开端，但对实际应用来说还远远不够。接下来我们对需要校验的 JSON 文档进行一些修改。

- 增加额外字段，如 `company`。
- 移除一个期望字段，如 `postedSlides`。

例 5-5 展示了修改后的 JSON 文档。

例 5-5 ex-2-basic-types-invalid.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "age": 39,
  "rating": 4.1,
  "company": "None"
}
```

运行命令后可以看到，经过以上修改，文档校验结果还是合法的。

```
json-at-work => validate ex-2-basic-types-invalid.json ex-2-basic-types-schema.json
JSON content in file ex-2-basic-types-invalid.json is valid
json-at-work =>
```

基础类型校验

至此，你可能会觉得 JSON Schema 没什么用处，毕竟上述校验命令并未如预期般工作。但通过增加一些简单的限制条件，就可以使校验过程像预期那样进行了。首先，可以像例 5-6 中的代码一样，禁止 JSON 中出现额外字段。

例 5-6 ex-3-basic-types-no-addl-props-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "postedSlides": {
      "type": "boolean"
    },
    "rating": {
      "type": "number"
    }
  },
  "additionalProperties": false
}
```

在以上示例中，将 `additionalProperties` 设置为 `false` 可以禁止 JSON 文档的根节点对象中出现额外字段。将之前的 JSON 文档（ex-2-basic-types-invalid.json）复制为一个新文件（ex-3-basic-types-no-addl-props-invalid.json），然后再使用之前的 Schema 进行校验，应该可以看到以下结果：

```
json-at-work => validate ex-3-basic-types-no-addl-props-invalid.json ex-3-basic-types-no-addl-props-schema.json
Invalid: Additional properties not allowed
JSON Schema element: /additionalProperties
JSON Content path: /age
```

与上一次校验的结果相比，这一结果好多了。然而，因为无法确保文档中包含所有的预期字段，所以这一结果也并不是我们最终想要的。为了在语义校验上达到核心水平，需要确保 JSON 中包含所有的必需字段，如例 5-7 所示。

例 5-7 ex-4-basic-types-validation-req-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string"
    },
    "firstName": {
```

```
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "postedSlides": {
      "type": "boolean"
    },
    "rating": {
      "type": "number"
    }
  },
  "additionalProperties": false,
  "required": ["email", "firstName", "lastName", "postedSlides", "rating"]
}
```

在以上示例中，`required` 数组声明所有的必需字段，JSON 文档则**必须**包含这些字段，否则即为非法。值得注意的是，如果某个字段不在 `required` 数组中，则意味着该字段属于可选字段。

例 5-8 展示了需要校验的修改后的 JSON 文档，该文档缺失必需的 `rating` 字段而多出了一个 `age` 字段。

例 5-8 ex-4-basic-types-validation-req-invalid.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "postedSlides": true,
  "age": 39
}
```

在命令行中运行后可以看到，该文档的校验结果为非法：

```
json-at-work => validate ex-4-basic-types-validation-req-invalid.json ex-4-basic-types-validation-req-schema.json
Invalid: Missing required property: rating
JSON Schema element: /required/4
JSON Content path:
```

最终，我们得到了预期的校验结果。

- 禁止出现额外的字段。
- Schema 中的所有字段都是必需的。

介绍了基本的语义校验后，我们来看一下 JSON 文档中的数值校验操作。

5.2.4　数值

你可能还记得，JSON Schema 中的 `number` 类型既可以表示浮点数，也可以表示整数。例 5-9 中的 Schema 校验了演讲者在会议报告方面的平均评分（`rating`），设定其范围为 1.0（极差）~5.0（优秀）。

例 5-9　ex-5-number-min-max-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "rating": {
      "type": "number",
      "minimum": 1.0,
      "maximum": 5.0
    }
  },
  "additionalProperties": false,
  "required": ["rating"]
}
```

根据该 Schema，例 5-10 中的 JSON 文档是合法的，因为其 `rating` 值在 1.0~5.0 这一范围内。

例 5-10　ex-5-number-min-max.json

```
{
  "rating": 4.99
}
```

例 5-11 中的 JSON 文档则是非法的，因为其 `rating` 值超过了 5.0。

例 5-11　ex-5-number-min-max-invalid.json

```
{
  "rating": 6.2
}
```

在命令行中校验例 5-11 中的文档后，可以看到校验结果为非法：

```
json-at-work => validate ex-5-number-min-max-invalid.json ex-5-number-min-max-schema.json
Invalid: Value 6.2 is greater than maximum 5
JSON Schema element: /properties/rating/maximum
JSON Content path: /rating
```

5.2.5　数组

可以使用 JSON Schema 来校验数组。数组中可以包含 JSON Schema 的任意基础类型（`string`、`number`、`array`、`object`、`boolean` 和 `null`）。例 5-12 中的 Schema 会校验 `tags` 字段，设定其为字符串数组。

例 5-12　ex-6-array-simple-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "tags": {
      "type": "array",
        "items": {
        "type": "string"
      }
    }
```

```
  },
  "additionalProperties": false,
  "required": ["tags"]
}
```

根据以上示例中的 Schema，例 5-13 中的 JSON 文档是合法的。

例 5-13 ex-6-array-simple.json

```
{
  "tags": ["fred"]
}
```

例 5-14 中的 JSON 文档则是非法的，因为其 `tags` 数组中包含整数。

例 5-14 ex-6-array-simple-invalid.json

```
{
  "tags": ["fred", 1]
}
```

在命令行中校验例 5-14 中的文档，可以看到校验结果为非法：

```
json-at-work => validate ex-6-array-simple-invalid.json ex-6-array-simple-schema.json
Invalid: invalid type: number (expected string)
JSON Schema element: /properties/tags/items/type
JSON Content path: /tags/1
```

JSON Schema 还可以校验数组成员的数目，设定最小数目（`minItem`）和最大数目（`maxItem`）。例 5-15 中的 Schema 会校验 `tags` 数组，设定其成员数为 2~4。

例 5-15 ex-7-array-min-max-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "tags": {
      "type": "array",
      "minItems": 2,
      "maxItems": 4,
      "items": {
        "type": "string"
      }
    }
  },
  "additionalProperties": false,
  "required": ["tags"]
}
```

根据该 Schema，例 5-16 中的 JSON 文档是合法的。

例 5-16 ex-7-array-min-max.json

```
{
  "tags": ["fred", "a"]
}
```

例 5-17 中的 JSON 文档则是非法的，因为其 `tags` 数组中包含了 5 个成员。

例 5-17　ex-7-array-min-max-invalid.json

```
{
  "tags": ["fred", "a", "x", "betty", "alpha"]
}
```

在命令行中进行校验，可以看到结果为非法：

```
json-at-work => validate ex-7-array-min-max-invalid.json ex-7-array-min-max-schema.json
Invalid: Array is too long (5), maximum 4
JSON Schema element: /properties/tags/maxItems
JSON Content path: /tags
```

5.2.6　枚举值

通过在数组中定义几个固定的枚举值，`enum` 关键词可以限制字段的取值。例 5-18 中的 Schema 限制了 `tags` 数组成员的值，限定其为 `"Open Source"`、`"Java"`、`"JavaScript"`、`"JSON"` 或 `"REST"` 中的一个。

例 5-18　ex-8-array-enum-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "tags": {
      "type": "array",
      "minItems": 2,
      "maxItems": 4,
      "items": {
        "enum": [
          "Open Source", "Java", "JavaScript", "JSON", "REST"
        ]
      }
    }
  },
  "additionalProperties": false,
  "required": ["tags"]
}
```

根据以上示例中的 Schema，例 5-19 中的 JSON 文档是合法的。

例 5-19　ex-8-array-enum.json

```
{
  "tags": ["Java", "REST"]
}
```

例 5-20 中的 JSON 文档则是非法的，因为其 `tags` 数组中包含的 `"JS"` 并不是 Schema 中的 `enum` 枚举值。

例 5-20　ex-8-array-enum-invalid.json

```
{
  "tags": ["Java", "REST", "JS"]
}
```

在命令行中进行校验后可以看到结果为非法：

```
json-at-work => validate ex-8-array-enum-invalid.json ex-8-array-enum-schema.json
Invalid: No enum match for: "JS"
JSON Schema element: /properties/tags/items/type
JSON Content path: /tags/2
```

5.2.7 对象

可以使用 JSON Schema 声明对象。利用这一点可以校验在应用程序间交换的对象数据，因此其是语义校验的核心。借助这一功能，API 的提供者和使用者可以就业务上一些重要概念（如 `person` 或 `order`）的结构和内容达成共识。例 5-21 中的 Schema 声明了 `speaker` 对象。

例 5-21 ex-9-named-object-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "speaker": {
      "type": "object",
      "properties": {
        "firstName": {
          "type": "string"
        },
        "lastName": {
          "type": "string"
        },
        "email": {
          "type": "string"
        },
        "postedSlides": {
          "type": "boolean"
        },
        "rating": {
          "type": "number"
        },
        "tags": {
          "type": "array",
          "items": {
            "type": "string"
          }
        }
      },
      "additionalProperties": false,
      "required": ["firstName", "lastName", "email",
        "postedSlides", "rating", "tags"
      ]
    }
  },
  "additionalProperties": false,
  "required": ["speaker"]
}
```

除了在根节点对象中增加了一级对象 `speaker`，这一 Schema 与之前示例中的基本类似。

根据该 Schema，例 5-22 中的 JSON 文档是合法的。

例 5-22　ex-9-named-object.json

```
{
  "speaker": {
    "firstName": "Larson",
    "lastName": "Richard",
    "email": "larsonrichard@ecratic.com",
    "postedSlides": true,
    "rating": 4.1,
    "tags": [
      "JavaScript", "AngularJS", "Yeoman"
    ]
  }
}
```

例 5-23 中的 JSON 文档则是非法的，因为其 `speaker` 对象中缺失必需的 `rating` 字段。

例 5-23　ex-9-named-object-invalid.json

```
{
  "speaker": {
    "firstName": "Larson",
    "lastName": "Richard",
    "email": "larsonrichard@ecratic.com",
    "postedSlides": true,
    "tags": [
      "JavaScript", "AngularJS", "Yeoman"
    ]
  }
}
```

在命令行中进行校验后，可以看到结果为非法：

```
json-at-work => validate ex-9-named-object-invalid.json ex-9-named-object-schema.json
Invalid: Missing required property: rating
JSON Schema element: /properties/speaker/required/4
JSON Content path: /speaker
```

至此，我们介绍了最重要的一些基础类型，接下来将介绍一些更复杂的 Schema。

5.2.8　模式属性

通过使用 `patternProperties` 关键词，JSON Schema 中的模式属性可以基于正则表达式来声明部分重复的字段名。例 5-24 定义了与地址相关的字段。

例 5-24　ex-10-pattern-properties-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "city": {
      "type": "string"
    },
    "state": {
```

```
        "type": "string"
      },
      "zip": {
        "type": "string"
      },
      "country": {
        "type": "string"
      }
    },
    "patternProperties": {
      "^line[1-3]$": {
        "type": "string"
      }
    },
    "additionalProperties": false,
    "required": ["city", "state", "zip", "country", "line1"]
  }
```

在以上示例中，正则表达式 `^line[1-3]$` 允许 JSON 文档中出现以下地址字段：`line1`、`line2` 和 `line3`。以下是对该正则表达式的一些解释。

- `^` 表示字符串开头。
- `line` 表示字符串 `"line"`。
- `[1-3]` 表示 1 至 3 之间的一个整数。
- `$` 表示字符串结尾。

值得注意的是，这 3 个地址字段中只有 `line1` 字段是必需的，其他字段都是可选的。

根据该 Schema，例 5-25 中的 JSON 文档是合法的。

例 5-25　ex-10-pattern-properties.json

```
{
  "line1": "555 Main Street",
  "line2": "#2",
  "city": "Denver",
  "state": "CO",
  "zip": "80231",
  "country": "USA"
}
```

例 5-26 中的 JSON 文档则是非法的，因为其中的 `line4` 字段并不在允许范围之内。

例 5-26　ex-10-pattern-properties-invalid.json

```
{
  "line1": "555 Main Street",
  "line4": "#2",
  "city": "Denver",
  "state": "CO",
  "zip": "80231",
  "country": "USA"
}
```

在命令行中进行校验后可以看到结果为非法：

```
json-at-work => validate ex-10-property-patterns-invalid.json ex-10-property-patterns-schema.json
Invalid: Additional properties not allowed
JSON Schema element: /additionalProperties
JSON Content path: /line4
```

5.2.9 正则表达式

JSON Schema 还可以使用正则表达式来限制字段值。例 5-27 中的 Schema 限制了 `email` 字段的值，限定其遵守 IETF RFC 2822 中标准的电子邮件地址格式。

例 5-27 ex-11-regex-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string",
      "pattern": "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    }
  },
  "additionalProperties": false,
  "required": ["email", "firstName", "lastName"]
}
```

在以上示例中，正则表达式声明了合法的电子邮件地址。以下是对该正则表达式的一些解释。

- `^` 表示字符串开头。
- `[\\w|-|.]+` 匹配以下模式的一对多实例：
  - `[\\w|-|.]` 匹配单词字符（`a-zA-Z0-9_`）、破折号（`-`）或点号（`.`）。
- `@` 表示字符“@”。
- `[\\w]+` 匹配以下模式的一对多实例：
  - `[\\w]` 匹配单词字符（`a-zA-Z0-9_`）。
- `\\.` 表示字符“.”。
- `[A-Za-z]{2,4}` 对 2~4 个以下模式进行匹配：
  - `[A-Za-z]` 匹配英文字符。
- `$` 表示字符串结尾。

JSON Schema 中的双反斜杠（`\\`）用于表示正则表达式中的特殊字符，这么做的原因是：JSON 文档的核心语法已经将单反斜杠用于转义特殊字符（如 `\b` 代表退格），因此标准正则表达式中的单反斜杠（`\`）无法在 JSON Schema 中正常工作。

根据该 Schema，例 5-28 中的 JSON 文档是合法的，因为 email 地址遵循了 Schema 中所声明的模式。

例 5-28　ex-11-regex.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard"
}
```

例 5-29 中的 JSON 文档则是非法的，因为其 email 地址缺失了 .com 后缀。

例 5-29　ex-11-regex-invalid.json

```
{
  "email": "larsonrichard@ecratic",
  "firstName": "Larson",
  "lastName": "Richard"
}
```

在命令行中进行校验后可以看到结果为非法：

```
json-at-work => validate ex-11-regex-invalid.json ex-11-regex-schema.json
Invalid: String does not match pattern: ^[\w|-]+@[\w|-]+\.[A-Za-z]{2,4}$
JSON Schema element: /properties/email/pattern
JSON Content path: /email
```

深入学习正则表达式

正则表达式有时显得复杂而又令人怯步。全面地介绍正则表达式超出了本书范围，如需掌握正则表达式，可以参考以下资源：

- Michael Fitzgerald 所著的 *Introducing Regular Expressions*（O'Reilly 出版社）；
- Jan Goyvaerts 和 Steven Levithan 所著的 *Regular Expression Cookbook, Second Edition*（O'Reilly 出版社）；
- Jeffrey E. F. Friedl 所著的《精通正则表达式（第 3 版）》；
- Regular Expression 101 网站，这是我最喜欢的正则表达式网站；
- RegExr 网站；
- Regular-Expressions.info 网站。

5.2.10　依赖属性

依赖属性在 Schema 中引入了依赖关系：某个字段依赖其他字段的存在。`dependencies` 关键词会定义包含依赖关系的对象，例如：只有某个数组内所有的字段都出现后，才可以使用 x 字段。在例 5-30 中，如果 JSON 文档内出现了 `favoriteTopics` 字段，则 `tags` 字段必须同时出现（即 `favoriteTopics` 字段依赖于 `tags` 字段）。

例 5-30　ex-12-dependent-properties-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string",
      "pattern": "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
    },
```

```
        "firstName": {
          "type": "string"
        },
        "lastName": {
          "type": "string"
            },
        "tags": {
          "type": "array",
            "items": {
              "type": "string"
            }
        },
        "favoriteTopic": {
          "type": "string"
        }
      },
      "additionalProperties": false,
      "required": ["email", "firstName", "lastName"],
      "dependencies": {
        "favoriteTopic": ["tags"]
      }
    }
```

根据以上示例中的 Schema，例 5-31 中的 JSON 文档是合法的，因为同时出现了 `favoriteTopics` 字段和 `tags` 数组。

例 5-31 ex-12-dependent-properties.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "tags": [
    "JavaScript", "AngularJS", "Yeoman"
  ],
  "favoriteTopic": "JavaScript"
}
```

例 5-32 中的 JSON 文档则是非法的，因为出现 `favoriteTopics` 字段时，`tags` 数组缺失。

例 5-32 ex-12-dependent-properties-invalid.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "favoriteTopic": "JavaScript"
}
```

在命令行中进行校验后可以看到结果为非法：

```
json-at-work => validate ex-12-dependent-properties-invalid.json ex-12-dependent-properties-schema.json
Invalid: Dependency failed - key must exist: tags (due to key: favoriteTopic)
JSON Schema element: /dependencies/favoriteTopic/0
JSON Content path:
```

5.2.11 内部引用

在 Schema 中使用引用可以重用定义 / 校验规则。可以将引用视作 Schema 领域内 DRY（Don't Repeat Yourself，避免重复代码）准则的一个实现。引用可以是内部的（引用当前 Schema 中的内容），也可以是外部的（引用外部 Schema 中的内容）。本节将介绍内部引用。

在例 5-33 中，可以看到 `email` 字段的正则表达式已经替换成 `$ref`，`$ref` 字段的值则是一个 URI，指向 `email` 字段真正的定义 / 校验规则。

- `#` 表示相关规则存在于当前 Schema 内部。
- `/definitions/` 表示当前 Schema 中 `definitions` 对象的路径。注意，此处 `definitions` 关键词表示使用的是一个引用。
- `emailPattern` 表示 `definitions` 对象中 `emailPattern` 规则的路径。
- JSON Schema 使用 JSON 指针（详见第 7 章）来声明 URI（如 `#/definitions/emailPattern`）。

例 5-33　ex-13-internal-ref-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "$ref": "#/definitions/emailPattern"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    }
  },
  "additionalProperties": false,
  "required": ["email", "firstName", "lastName"],
  "definitions": {
    "emailPattern": {
      "type": "string",
      "pattern": "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
    }
  }
}
```

除了新增的 `definitions` 对象，以上示例与之前的 Schema 并无多少不同。例 5-33 只是将电子邮件地址的相关规则挪到一处公用的地方，使得 Schema 内的多个字段可以共享该规则而已。

根据该 Schema，例 5-34 中的 JSON 文档是合法的。

例 5-34　ex-13-internal-ref.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard"
}
```

例 5-35 中的 JSON 文档则是非法的，因为其 `email` 地址缺失了 `.com` 后缀。

例 5-35　ex-13-internal-ref-invalid.json

```
{
  "email": "larsonrichard@ecratic",
  "firstName": "Larson",
  "lastName": "Richard"
}
```

在命令行中进行校验后可以看到结果为非法：

```
json-at-work => validate ex-13-internal-ref-invalid.json ex-13-internal-ref-schema.json
Invalid: String does not match pattern: ^[\w|-]+@[\w|-]+\.[A-Za-z]{2,4}$
JSON Schema element: /properties/email/pattern
JSON Content path: /email
```

5.2.12　外部引用

外部引用用于声明位于外部 Schema 文件中的校验规则。在这种情况下，Schema A 可以引用 Schema B 中的一些特定校验规则。外部引用使得企业内的开发团队（或多个开发团队间）可以重用常用的 Schema。

例 5-36 展示了引用外部 Schema 的演讲者数据 Schema。

例 5-36　ex-14-exernal-ref-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "$ref":
        "http://localhost:8081/ex-14-my-common-schema.json#/definitions/emailPattern"
      },
      "firstName": {
        "type": "string"
      },
      "lastName": {
        "type": "string"
      }
  },
  "additionalProperties": false,
  "required": ["email", "firstName", "lastName"]
}
```

与例 5-33 中的 Schema 相比，这一 Schema 存在两个关键性的区别。

- 该 Schema 移除了 `definitions` 对象。不用担心，我们很快就会再次看到它。
- `email` 字段的 `$ref` 如今指向了外部 Schema 文件（ex-14-my-common-schema.json），在该外部文件中搜索需要的定义 / 校验规则。本章稍后将介绍外部 Schema 的 HTTP 地址。

例 5-37 展示了该外部 Schema 的内容。

例 5-37　ex-14-my-common-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://localhost:8081/ex-14-my-common-schema.json",

  "definitions": {
    "emailPattern": {
      "type": "string",
      "pattern": "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
    }
  }
}
```

包含 `emailPattern` 校验规则的 `definitions` 对象如今保存在该外部 Schema 文件中。你可能会有以下困惑。

- 引用到底是如何工作的？
- JSON Schema 校验器如何定位外部 Schema？

以下是上述问题的答案。

- 在 ex-14-external-ref-schema.json 文件中，`$ref` 中 # 前的 URI 前缀（http://localhost:8081/ex-14-my-common-schema.json）表示 JSON Schema 处理器应该在外部 Schema 中搜索 `emailPattern` 定义。
- 在外部 Schema 文件 ex-14-my-common-schema.json 中，Schema 根节点中的 `id` 字段（JSON Schema 的关键词之一）声明此 Schema 内容可由外部文件引用。
- `$ref` 中的 URI 和 `id` 字段的值必须严格匹配才能让引用正常工作。
- `definitions` 对象的工作方式与内部引用相同。

根据该 Schema，例 5-38 中的 JSON 文档是合法的。值得注意的是，与之前的 JSON 文档相比，该 JSON 没有任何修改，也不会意识到外部 Schema 的存在。

例 5-38　ex-14-external-ref.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard"
}
```

例 5-39 中的 JSON 文档则是非法的，因为其 `email` 地址缺失了 `.com` 后缀。

例 5-39　ex-14-external-ref-invalid.json

```
{
  "email": "larsonrichard@ecratic",
  "firstName": "Larson",
  "lastName": "Richard"
}
```

使用 Schema 校验这一文档的方式有两种：

- 通过文件系统实现外部引用；
- 通过 Web 实现外部引用。

首先，我们来看一下用 `validate` 工具在文件系统上实现的外部引用校验：

```
json-at-work => validate ex-14-external-ref-invalid.json ex-14-external-ref-schema.json
Invalid: String does not match pattern: ^[\w|-]+@[\w|-]+\.[A-Za-z]{2,4}$
JSON Schema element: /properties/email/pattern
JSON Content path: /email
```

运行结果显示 ex-14-external-ref-invalid.json 这一 JSON 文档非法，不过该示例的重点在于：`validate` 命令在执行时既包含了主 Schema 文件（ex-14-external-ref-schema.json），也包含了外部引用的 Schema 文件（ex-14-my-common-schema.json）。

接下来，我们看一下在 Web 上实现的外部引用校验。我们将 Schema 文件以静态资源的形式部署到 Web 服务器中，从而使得 `$ref` 和 `id` 中的 URI（http://localhost:8081/ex-14-my-common-schema.json#/definitions/emailPattern）可以正常工作。如果尚未安装 Node.js 中的 `http-server` 模块，那么可以参考 A.2.5 节中的内容，在机器上安装并运行这一模块。

在外部 Schema 文件所在的路径下以 8081 端口运行 `http-server`，可以看到以下结果：

```
json-at-work => http-server -p 8081
Starting up http-server, serving ./ on: http://0.0.0.0:8081
Hit CTRL-C to stop the server
[Wed, 02 Sep 2015 03:44:00 GMT] "GET /ex-14-my-common-schema.json" "undefined"
[Wed, 02 Sep 2015 03:44:16 GMT] "GET /ex-14-my-common-schema.json" "undefined"
```

当在浏览器中访问 http://localhost:8081/ex-14-my-common-schema.json 时，可以看到如图 5-5 所示的结果。

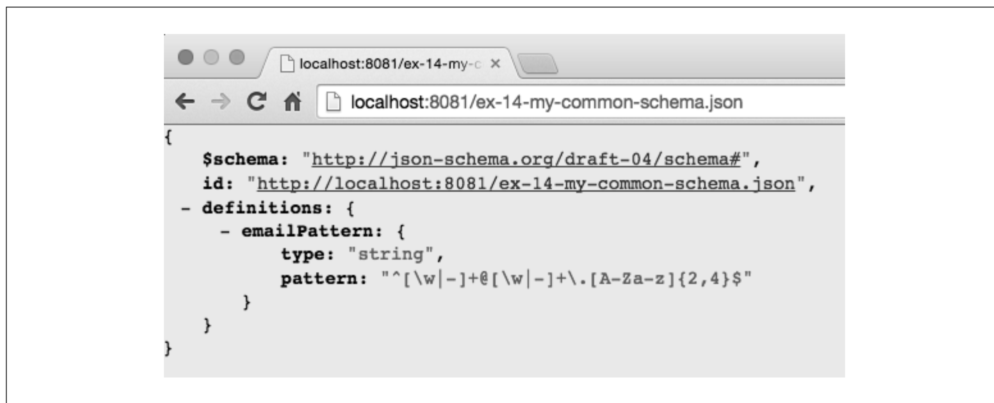


图 5-5：可通过 Web 访问的外部 Schema

对外部 Schema 开放 Web 访问后，我们执行校验操作，然后就可以看到文档的校验结果为非法：

```
json-at-work => validate ex-14-external-ref-invalid.json ex-14-external-ref-schema.json
Invalid: String does not match pattern: ^[\w|-]+@[\w|-]+\.[A-Za-z]{2,4}$
JSON Schema element: /properties/email/pattern
JSON Content path: /email
```

5.2.13　选择校验规则

除了 `requires` 和 `dependencies` 关键词，对于供 Schema 处理器使用的校验规则，JSON Schema 还提供更多细粒度的机制。以下是一些相关的关键词。

`oneOf`
: 有且仅有一条规则匹配。

`anyOf`
: 一条或多条规则匹配。

`allOf`
: 所有规则均匹配。

1. `oneOf`

`oneOf` 关键词在多条校验规则中强制进行排他选择（有且仅有一条规则匹配）。在例 5-40 的 Schema 中，`rating` 字段的值可以小于 2.0，或者大于 5.0，但不能同时满足这两点。

例 5-40　ex-15-one-of-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string",
      "pattern": "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
    },
    "firstName": {
      "type": "string"
    },
    "postedSlides": {
      "type": "boolean"
    },
    "rating": {
      "type": "number",
      "oneOf": [
        {
          "maximum": 2.0
        },
        {
          "minimum": 5.0
        }
      ]
    }
  },
  "additionalProperties": false,
  "required": [ "email", "firstName", "lastName", "postedSlides", "rating" ]
}
```

根据该 Schema，例 5-41 中的 JSON 文档是合法的，因为 `rating` 字段的值为 4.1，仅与其中一条校验规则匹配（<5.0）。

例 5-41　ex-15-one-of.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "postedSlides": true,
  "rating": 4.1
}
```

例 5-42 中的 JSON 文档则是非法的，因为其 `rating` 字段的值 1.9 同时与两条校验规则相匹配（<2.0 和 <5.0）。

例 5-42　ex-15-one-of-invalid.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "postedSlides": true,
  "rating": 1.9
}
```

在命令行中进行校验后可以看到结果为非法：

```
json-at-work => validate ex-15-one-of-invalid.json ex-15-one-of-schema.json
Invalid: Data is valid against more than one schema from "oneOf": indices 0 and 1
JSON Schema element: /properties/rating/oneOf
JSON Content path: /rating
```

2. anyOf

`anyOf` 关键词在多条校验规则中检测是否出现匹配。在例 5-43 中，除了布尔值，`postedSlides` 字段还支持 `[Y|y]es` 和 `[N|n]o` 字符串。

例 5-43　ex-16-any-of-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
      "type": "string",
      "pattern": "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "postedSlides": {
      "anyOf": [
        {
          "type": "boolean"
        },
        {
```

```
          "type": "string",
          "enum": [ "yes", "Yes", "no", "No" ]
        }
      ]
    },
    "rating": {
      "type": "number"
    }
  },
  "additionalProperties": false,
  "required": [ "email", "firstName", "lastName", "postedSlides", "rating" ]
}
```

根据该 Schema，例 5-44 中的 JSON 文档是合法的，因为 `postedSlides` 字段的值为 `"yes"`。

例 5-44 ex-16-any-of.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "postedSlides": "yes",
  "rating": 4.1
}
```

例 5-45 中的 JSON 文档则是非法的，因为其 `postedSlides` 字段的值为 `"maybe"`，但允许的值中并不包含 `"maybe"`。

例 5-45 ex-16-any-of-invalid.json

```
{
  "email": "larsonrichard@ecratic.com",
  "firstName": "Larson",
  "lastName": "Richard",
  "postedSlides": "maybe",
  "rating": 4.1
}
```

在命令行中进行校验后可以看到结果为非法：

```
json-at-work => validate ex-16-any-of-invalid.json ex-16-any-of-schema.json
Invalid: Data does not match any schemas from "anyOf"
JSON Schema element: /properties/postedSlides/anyOf
JSON Content path: /postedSlides
```

3. allOf

使用 `allOf` 关键词后，数据必须与所有的规则都匹配。在例 5-46 中，`lastName` 字段**必须**是长度小于 20 的字符串。

例 5-46 ex-17-all-of-schema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "email": {
```

```
        "type": "string",
        "pattern": "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
      },
      "firstName": {
        "type": "string"
      },
      "lastName": {
        "allOf": [
          { "type": "string" },
          { "maxLength": 20 }
        ]
      },
      "postedSlides": {
        "type": "boolean"
      },
      "rating": {
        "type": "number",
        "maximum": 5.0
      }
    },
    "additionalProperties": false,
    "required": [
      "email",
      "firstName",
      "lastName",
      "postedSlides",
      "rating"
    ]
  }
```

根据该 Schema，例 5-47 中的 JSON 文档是合法的，因为 `lastName` 字段的长度≤ 20。

例 5-47 ex-17-all-of.json

```
  {
    "email": "larsonrichard@ecratic.com",
    "firstName": "Larson",
    "lastName": "Richard",
    "postedSlides": true,
    "rating": 4.1
  }
```

例 5-48 中的 JSON 文档则是非法的，因为其 `lastName` 字段的长度超过了 20 个字符。

例 5-48 ex-17-all-of-invalid.json

```
  {
    "email": "larsonrichard@ecratic.com",
    "firstName": "Larson",
    "lastName": "ThisLastNameIsWayTooLong",
    "postedSlides": true,
    "rating": 4.1
  }
```

在命令行中进行校验后可以看到结果为非法：

```
json-at-work => validate ex-17-all-of-invalid.json ex-17-all-of-schema.json
Invalid: String is too long (24 chars), maximum 20
JSON Schema element: /properties/lastName/allOf/1/maxLength
JSON Content path: /lastName
```

对 JSON Schema 及其语法进行了基本介绍后，接下来我们探讨如何使用 JSON Schema 来设计 API。

5.3 如何使用JSON Schema设计和测试API

JSON Schema 的意义在于定义应用程序和 API 间数据交换的语义和结构。在 API 设计领域，可以将 JSON Schema 理解为协议（接口）的一部分。本节将基于理论概念创建一个可运行的模拟 API，且其他应用程序和 API 可以对此模拟 API 进行测试与使用。

5.3.1 应用场景

我们将继续使用之前章节中的演讲者数据模型，并逐步添加约束和功能。以下是从概念到可运行的模拟 API 所需要执行的步骤。

(1) 对 JSON 文档进行建模。
(2) 生成 JSON Schema。
(3) 生成示例数据。
(4) 用 `json-server` 部署模拟 API。

5.3.2 JSON文档建模

在创建 Schema 前，我们需要先了解交换的数据。除了数据的字段及格式外，对数据本身的样子有个良好的感性认知也是很重要的。要想获取良好的感性认知，最大的困难来自于 JSON 本身：手动创建 JSON 文档繁琐且易错。因此，应当使用建模工具来避免大量的手动输入工作。在诸多优秀的工具中，我最中意的是 JSON Editor Online。有关 JSON Editor Online 的功能，参见 1.5.3 节。

图 5-6 展示了 JSON Editor Online 中的 `speaker` 模型。

在创建 JSON 文档时，应当先使用 JSON Editor Online 对数据进行建模，然后生成 JSON 文档，而不是直接手动输入 JSON 文档。在图 5-6 右侧区域的 JSON 模型中，点击对象、名称－值对或数组等元素旁的图标，即可在出现的目录中选择添加 / 插入以下新的元素：

- 对象
- 名称－值对
- 数组



图 5-6：jsoneditoronline.com 网站中的演讲者数据模型

添加一些字段后，可以点击页面中间的左箭头按钮来生成 JSON 文档。接下来就可以对文档内容进行迭代式的添加、测试和检查，直到满意为止。最后，通过选择 Save 目录下的 Save to Disk 选项，将 JSON 文档保存为本地文件，如例 5-49 所示。

例 5-49　ex-18-speaker.json

```
{
  "about": "Fred Smith is the CTO of Full Ventures, where he ...",
  "email": "fred.smith@fullventures.com",
  "firstName": "Fred",
  "lastName": "Smith",
  "picture": "http://placehold.it/fsmith-full-ventures-small.png",
  "tags": [
    "JavaScript",
    "REST",
    "JSON"
  ],
  "company": "Full Ventures, Inc."
}
```

在继续深入前，最好先使用 JSONLint（Web 应用程序或命令行界面）来校验刚生成的 JSON 文档。虽然 JSON Editor Online 所生成的 JSON 都是合法的，但进行一下复查总归不会有错。

5.3.3　生成JSON Schema

生成合法的 JSON 文档后，即可根据该文档的结构与内容使用 JSONSchema.net 来生成相应的 JSON Schema。再次强调，使用工具能够节省大量的手动输入工作。

访问图 5-7 所示网站并将 JSON 文档粘贴在左侧输入框中。

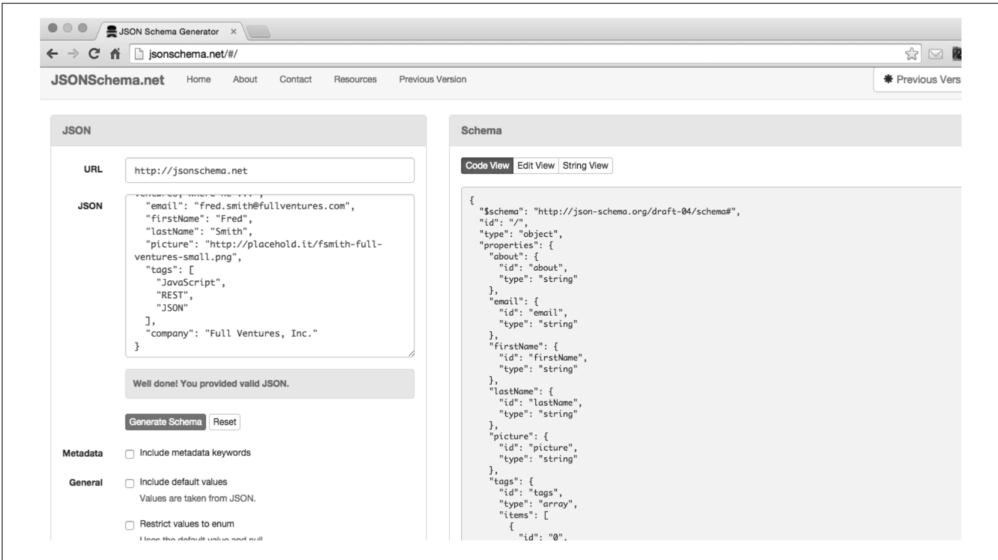


图 5-7：在 JSONSchema.net 网站上生成演讲者数据 Schema

使用默认设置，并进行以下变更来生成 Schema。

- 关闭“Use absolute IDs”。
- 关闭“Allow additional properties”。
- 点击 Generate Schema 按钮。
- 将右侧区域中生成的 Schema 复制到粘贴板中。

将粘贴板中的 Schema 保存到本地文件后，就可以看到如例 5-50 所示的结果。

例 5-50　ex-18-speaker-schema-generated.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "/",
  "type": "object",
  "properties": {
    "about": {
      "id": "about",
      "type": "string"
    },
    "email": {
      "id": "email",
      "type": "string"
    },
    "firstName": {
      "id": "firstName",
      "type": "string"
    },
    "lastName": {
      "id": "lastName",
```

```
        "type": "string"
      },
      "picture": {
        "id": "picture",
        "type": "string"
      },
      "tags": {
        "id": "tags",
        "type": "array",
        "items": [{
          "id": "0",
          "type": "string"
        }, {
          "id": "1",
          "type": "string"
        },{
          "id": "2",
          "type": "string"
        }]
      },
      "company": {
        "id": "company",
        "type": "string"
      }
    },
    "additionalProperties": false,
    "required": [
      "about",
      "email",
      "firstName",
      "lastName",
      "picture",
      "tags",
      "company"
    ]
  }
```

JSONSchema.net 在生成基本的 Schema 方面是非常不错的。遗憾的是，该工具会添加一些我们不需要的字段，同时也不支持 `enum`、`pattern` 等高级特性。使用 JSONSchema.net 的意义在于它可以完成 80% 的工作，剩下的工作则交由你自行完善。目前我们还不需要 `id` 字段，但需要使用正则表达式来校验 `email` 字段（与之前的示例一样）。完成相关的修改工作后，新的 Schema 如例 5-51 所示。

例 5-51 ex-18-speaker-schema-generated-modified.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "about": {
      "type": "string"
    },
    "email": {
      "type": "string",
      "pattern": "^[\\w|-|.]+@[\\w]+\\.[A-Za-z]{2,4}$"
    },
    "firstName": {
```

```
        "type": "string"
      },
      "lastName": {
        "type": "string"
      },
      "picture": {
        "type": "string"
      },
      "tags": {
        "type": "array",
        "items": [
          {
            "type": "string"
          }
        ]
      },
      "company": {
        "type": "string"
      }
    },
    "additionalProperties": false,
    "required": [ "about", "email", "firstName",
                  "lastName", "picture", "tags", "company"
    ]
}
```

5.3.4　校验JSON文档

生成 JSON Schema 后，接下来我们使用 JSON Validate 这一 Web 应用程序来校验 JSON 文档。访问图 5-8 所示网站并将 JSON 文档和 Schema 粘贴到相应的区域。

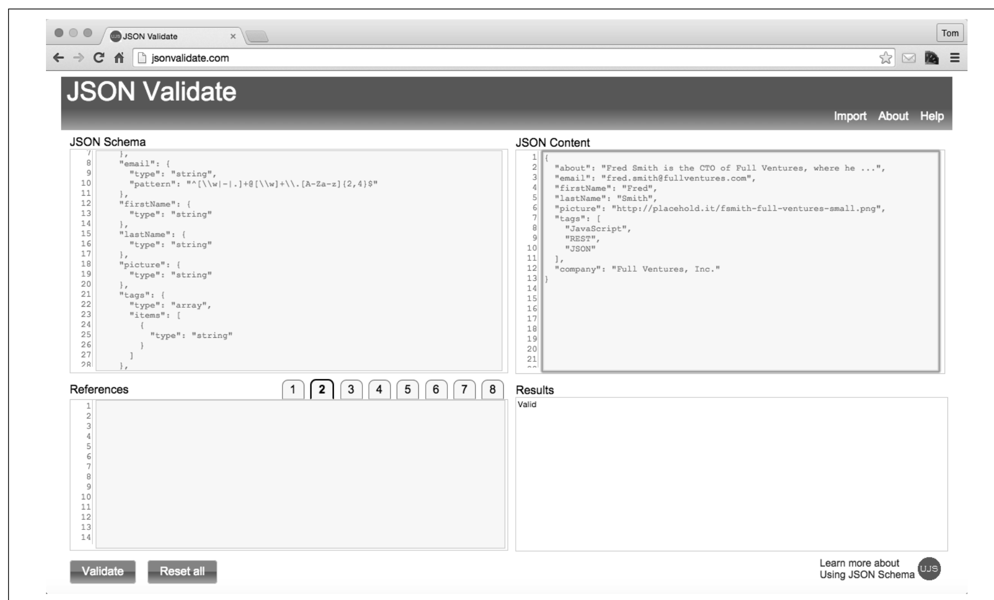


图 5-8：在 jsonvalidate.com 网站上使用 JSON Schema 来校验演讲者数据 JSON 文档

点击 Validate 按钮，对于该 Schema 来说，JSON 文档的校验结果应该是合法的。你可能会使用本章中提及的 `validate` 命令行界面工具，但 Web 应用程序在可视化方面做得更好。

5.3.5 生成示例数据

至此，我们已经拥有了一份 JSON 文档，也准备好了相应的 Schema，但对于创建测试 API 来说，还需要更多的数据。可以使用 JSON Editor Online 来生成测试数据，但这一方案存在一些纰漏：真实性强的测试数据对数据量和数据的随机性有较高要求。即使使用 GUI，这一目标也会消耗大量人力。

JSON Editor Online 更适用于创建小型的 JSON 文档，以推进项目设计；但我们需要进行 API 测试，因此需要能够产生大量随机 JSON 数据的方案。本节会使用 JSON 生成器来生成数据。访问图 5-9 所示网站后，即可看到显示结果。

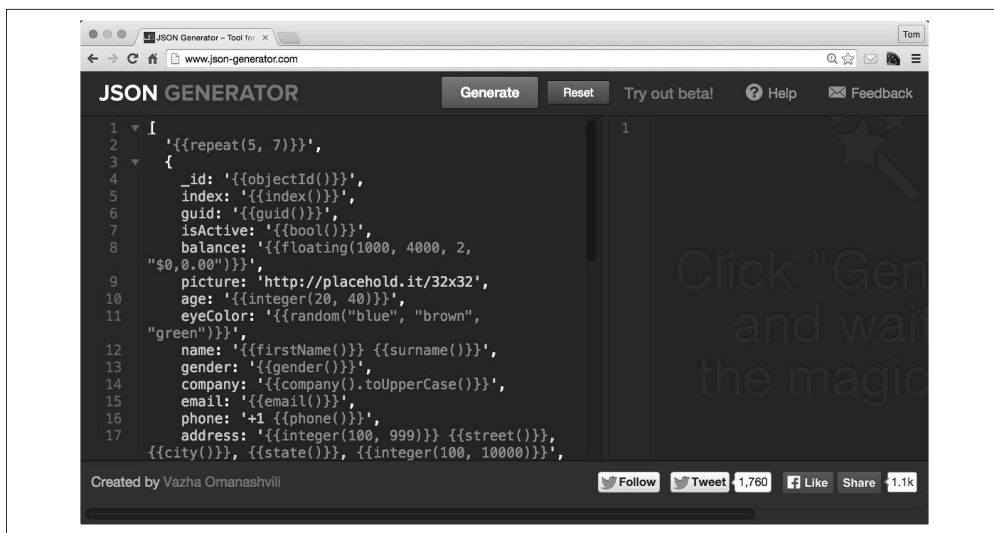


图 5-9：json-generator 网站

网站左侧区域的代码是一个模板，以 JavaScript 对象字面量的形式表示，用于生成示例 JSON 数据。值得注意的是，JSON 生成器可以生成示例 / 随机的段落、数值、名称、全局唯一标识符、性别、电子邮件地址等。另外，还可以在模板顶部使用 `{{repeat}}` 标签来批量生成数据。如需了解有关标签的详细文档，可点击 Help 按钮。

不过，模板中这么多的默认设置已经远远超过了我们的需求。例 5-52 对模板内容进行了删减，只留下所需要的字段来生成包含随机数据的 3 个 `speaker` 对象。

例 5-52　ex-18-speaker-template.js

```
// http://www.json-generator.com/ 中所使用的模板

[
  '{{repeat(3)}}', {
    id: '{{integer()}}',
```

```
    picture: 'http://placehold.it/32x32',
    name: '{{firstName()}}',
    lastName: '{{surname()}}',
    company: '{{company()}}',
    email: '{{email()}}',
    about: '{{lorem(1, "paragraphs")}}'
  }
]
```

点击 Generate 按钮后，在 Web 应用程序中可以看到如图 5-10 所示的 JSON 文档（如果需要更多的 `speaker` 对象，将 `repeat` 标签中的 3 改为更大的数字即可）。

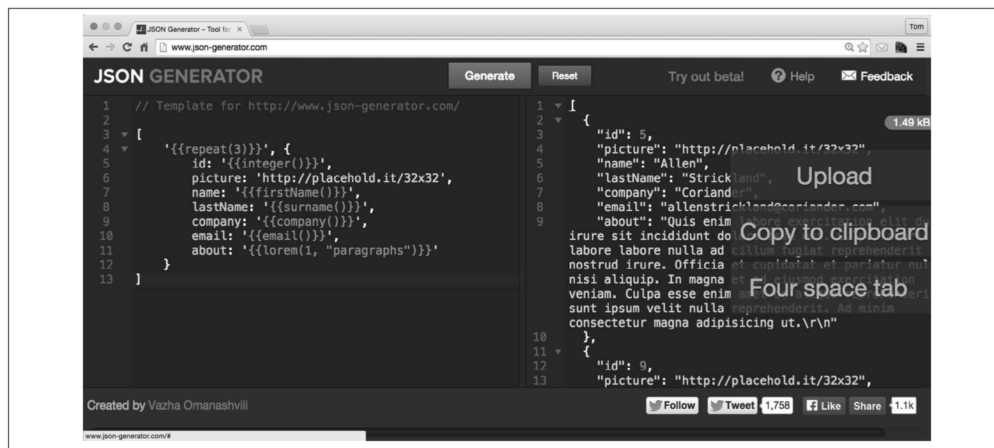


图 5-10：用 json-generator 生成演讲者数据 JSON 文档

现在，在右侧区域中点击 Copy to Clipboard 按钮，将 JSON 内容复制到文件中，如例 5-53 所示。

例 5-53　ex-18-speakers-generated.json

```
[
  {
    "id": 5,
    "picture": "http://placehold.it/32x32",
    "name": "Allen",
    "lastName": "Strickland",
    "company": "Coriander",
    "email": "allenstrickland@coriander.com",
    "about": "Quis enim labore ..."
  },
  {
    "id": 9,
    "picture": "http://placehold.it/32x32",
    "name": "Merle",
    "lastName": "Prince",
    "company": "Xylar",
    "email": "merleprince@xylar.com",
    "about": "Id voluptate duis ..."
  },
```

```
  {
    "id": 8,
    "picture": "http://placehold.it/32x32",
    "name": "Salazar",
    "lastName": "Ewing",
    "company": "Zentime",
    "email": "salazarewing@zentime.com",
    "about": "Officia qui id ..."
  }
]
```

至此，JSON 文档的创建已经接近完成。不过，为了将该文件部署为 API，我们需要对数据进行一些修改。

- JSON 文档中包含了一个数组。将该数组命名为 `speakers`，并用 `{` 和 `}` 括起来。经过这一修改后，JSON 文档的根节点元素会包含名为 `speakers` 的数组。
- 重新添加 `id` 字段，以 0 开始。

最终的 JSON 文件如例 5-54 所示。

例 5-54　ex-18-speakers-generated-modified.json

```
{
  "speakers": [
    {
      "id": 0,
      "picture": "http://placehold.it/32x32",
      "name": "Allen",
      "lastName": "Strickland",
      "company": "Coriander",
      "email": "allenstrickland@coriander.com",
      "about": "Quis enim labore ..."
    },
    {
      "id": 1,
      "picture": "http://placehold.it/32x32",
      "name": "Merle",
      "lastName": "Prince",
      "company": "Xylar",
      "email": "merleprince@xylar.com",
      "about": "Id voluptate duis ..."
    },
    {
      "id": 2,
      "picture": "http://placehold.it/32x32",
      "name": "Salazar",
      "lastName": "Ewing",
      "company": "Zentime",
      "email": "salazarewing@zentime.com",
      "about": "Officia qui id ..."
    }
  ]
}
```

你可能会对上述修改的用意感到有些困惑。这么做的目的在于：完成修改后，`json-server` 能够以合适的 URI 提供演讲者数据。

- 通过将数组封装到 `speakers` 中，产生的 http://localhost:5000/speakers 路由能够以地址化的方式提供所有数据。
- 可以通过 http://localhost:5000/speakers/0 路由访问第一个元素。

我们已经有些超前了。接下来本节会运行 `json-server` 并浏览 API。

5.3.6 用`json-server`部署模拟API

准备好 Schema 和测试数据后，就可以将示例数据部署为 API，以便 API 的使用者开始测试并提供反馈。如果尚未安装 `json-server` 这一 Node.js 模块，那么可以参考 A.2.5 节中的相关指导步骤，安装并运行 `json-server`。

在 ex-18-speakers- generated-modified.json 文件所在的路径下，以 5000 端口运行 `json-server`，可以看到以下结果：

```
json-at-work => json-server -p 5000 ./ex-18-speakers-generated-modified.json
{^_^} Hi!

Loading database from ./ex-18-speakers-generated-modified.json
  http://localhost:5000/speakers

You can now go to http://localhost:5000/

Enter `s` at any time to create a snapshot of the db

GET /speakers 200 11.750 ms - 1667
GET /speakers 304 4.027 ms - -
GET /speakers/0 200 3.170 ms - 574
GET /speakers/0 304 2.556 ms - -
GET /speakers/0 304 1.350 ms - -
GET /speakers 304 1.437 ms - -
```

在浏览器中访问 http://localhost:5000/speakers，可以看到如图 5-11 所示的结果。

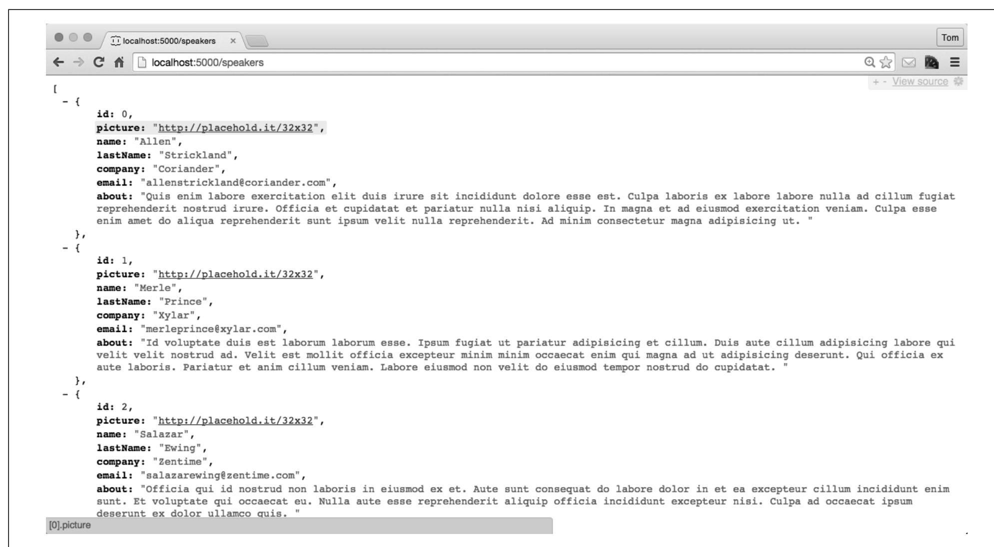


图 5-11：`json-server` 提供的演讲者数据模拟 API

至此，在没有编写任何代码的情况下，我们只通过部署静态 JSON 文件就搭建了一个可测试的 API。这一方案的优雅之处在于：无论从外观、功能还是使用感受上来说，效果和真正的 API 并无二致。该 API 所提供的交互操作也和其他 API 没有什么不同。你可以通过浏览器、cURL 或者在编程语言中发送 HTTP 请求的方式来使用该 API。

不过，使用 `json-server` 会有一个限制：只能对数据发送 HTTP `GET` 请求，因为接口是只读的。

5.3.7 关于用JSON Schema设计和测试API的一些思考

介绍完所有示例后，利用强大的开源 JSON 工具来缩短 API 开发周期这一点应该会给你留下深刻的印象。以下是总结。

- 在最终确认数据结构前，使用 JSON 建模工具进行设计。尽早与相关负责人进行经常性沟通。
- 手动编写 JSON 文档或 Schema 文件繁琐且易错。应当使用工具来完成大部分工作，尽量避免人工操作。
- 及早地、经常性地校验文档。
- 使用工具生成大量的随机 JSON 数据，而不是自己手动创建。
- 搭建模拟 API 是一件比较简单的事情。因为已经有现成的测试基础架构了，所以不要试图从头编写。直接使用现成的工具，你的时间应该用来做更有价值的事情。

5.4 使用JSON Schema类库进行校验

本章前面介绍了如何使用 `validate` 命令行工具以及 JSON Validate Web 应用程序对 JSON 文档进行 Schema 校验，而我们的终极目标是在应用程序内部实现校验操作。

不过，JSON Schema 并不仅仅适用于 JavaScript 和 Node.js。对于 JSON Schema v4，大多数主流编程平台都提供了极好的支持。

Ruby on Rails
: `json-schema` gem。

Java
: `json-schema-validator`。

PHP
: `jsv4-php`。

Python
: `jsonschema`。

Clojure
: 直接使用 Java 中的 `json-schema-validator` 即可。

Node.js
: Node.js 中存在不少高质量的 JSON Schema 处理器。我成功使用过的工具有以下两个。

- `ajv` 是我在 Node.js 应用程序中最喜欢使用的类库，因为它很简洁，可以与 Node.js 中流行的测试框架（Mocha/Chai、Jasmine 和 Karma 等）兼容。可以在 npm 网站和 GitHub 上找到有关 `ajv` 的更多信息。第 10 章将详细介绍 `ajv` 的用法。
- `ujs-jsonvalidate` 处理器是本章前面进行命令行校验时一直使用的工具。可以在其 GitHub 页面上找到更多信息。`ujs-jsonvalidate` 的 npm 模块地址为 https://www.npmjs.com/package/ujs-jsonvalidate。

5.5 如何继续深入学习JSON Schema

本章描述了 JSON Schema 的一些基本知识，但全面的相关介绍已超出了本章的范围。除了之前提到的 json-schema.org 网站，以下是其他一些相关资源。

- Joe McIntyre 发起的 Using JSON Schema 项目提供了丰富的 JSON Schema 信息及相关工具，其中包括：
  - *Using JSON Schema* 电子书；
  - jsonvalidate 应用程序；
  - `ujs-validate` npm 模块。
- Michael Droettboom 等所著的 *Understanding JSON Schema*。
- 关于 JSON Schema 的一个简介。

5.6 本章回顾

本章介绍了 JSON Schema 以及它在应用程序架构中的作用，然后使用 JSON Schema 及相关工具对 API 进行了设计和测试。

5.7 内容预告

了解了如何使用 JSON Schema 来结构化及校验 JSON 文档后，第 6 章将介绍如何搜索 JSON 文档。

第 6 章

在JSON中进行搜索

通过使用 JSON 的搜索类库和工具，可以简化 JSON 文档的搜索操作，并快速访问需要查找的字段。尤其在 Web API 返回的大型 JSON 文档中搜索内容时，JSON 搜索可以大显身手。

本章将介绍以下内容：

- 使用 JSON 搜索来简化工作；
- 使用主流的 JSON 搜索类库与工具；
- 编写单元测试，在 Web API 返回的 JSON 文档中进行搜索。

在本章示例中，我们将使用多种 JSON 搜索技术来搜索由本机 Web API 所提供的 JSON 数据。除此之外，我们还将创建单元测试，从而在测试中执行搜索操作并检查搜索结果。

6.1 为什么要在JSON中进行搜索

设想以下场景：某次 API 调用返回了几百个（或更多）JSON 对象，而你只需要其中的一部分数据（名称 – 值对），或者需要根据某个标准来搜索过滤返回的内容。如果没有 JSON 搜索操作，你就不得不自己编写代码来解析 JSON 文档，并在庞大的数据结构中进行遍历计算。这种偏底层的解决方案很繁琐，同时也意味着庞杂的处理代码。你的时间应该用来做更有价值的事情。本章介绍的 JSON 搜索工具可以有效降低你的工作量，并简化工作难度。

6.2 JSON搜索类库和工具

（可在应用程序内调用的）很多类库和命令行工具都可以用于搜索 JSON 文档。以下是本章

将介绍的一些广为使用的类库：

- JSONPath
- JSON Pointer
- `jq`

6.2.1　其他优秀工具

很多高质量的类库和命令行工具都可以用于搜索和过滤 JSON 内容，但本书囿于篇幅无法一一介绍。以下是其他一些值得了解的工具，为简洁起见，本章后面不再对它们进行详细讨论。

SpahQL
　SpahQL 有点像是以 JSON 对象为目标的 jQuery，可以在其 GitHub 主页上找到该类库。

`json`
　可以在 GitHub 和 npm 网站上找到该命令行工具。虽然本章不会使用 `json` 的搜索功能，但依旧会用它来优化 JSON 文档的显示。

`jsawk`
　`jsawk` 是一个命令行工具，除搜索外，它还支持 JSON 文档的转换操作。

虽然本书不会详细介绍这些工具的使用，但在你的项目中可能会出现适用上述工具的场景。可以将这些工具与 JSONPath、JSON Pointer 和 `jq` 进行对比，以便确定最佳选择。

6.2.2　选择工具的标准

因为存在大量的类库和工具，所以作出使用选择还是比较困难的。以下是我的具体选择标准。

关注度
　该工具是否广泛使用？当在网络上进行相关搜索时，能出来多少条搜索结果？

开发者社区
　代码是不是在 GitHub 上？维护的状态怎么样？

平台
　是否能跨平台工作？标准或类库接口是否得到了广泛支持？

易于入门
　文档是否齐全？安装难度如何？接口设计是否符合直觉？使用难度如何？

标准
　是否存在相关标准（如 IETF、W3C 或 Ecma 等）？

本章将使用上述标准来评估 JSON 搜索工具。

6.3 测试数据

为了演示搜索操作，我们需要更真实的测试数据和更大、更复杂的 JSON 文档，网络上存在大量的类似资源。在本章和下一章中，我们不再使用之前章节中的演讲者数据，而是从公用 API 中获取开放数据。我们将使用 OpenWeatherMap API 所提供的城市 / 天气数据。可在其官方网站上查看具体的 API 文档。

本书附带的 chapter-6/data/cities-weather-orig.json 文件中包含了 OpenWeatherMap API 所提供的美国加州南部城市（以经纬度划分的一个矩形区域）的天气数据。需要注意的是，OpenWeatherMap 所提供的天气数据会发生频繁的变化，因此本书示例中所截取的数据与该 API 所提供的最新数据会有一些出入。将天气数据部署到 `json-server` 前，我们会先对其进行一些修改。首先，可以在 data/cities-weather-orig.json 文件中看到天气数据保存在一个名为 `list` 的数组中。为了明确并增加可测试性，我们将该数组重命名为 `cities`，并将改动保存到 data/cities-weather.json 文件中。另外，我们将 JSON 文档头部的 `cod`、`calctime` 和 `cnt` 字段移到一个新的对象中，使之与 `json-server` 兼容，因为 `json-server` 只支持对象或对象数组。与之前的章节一样，我们会继续使用 `json-server` 这一 Node.js 模块将城市天气数据部署为 Web API。例 6-1 展示了修改后的天气数据。

例 6-1 data/cities-weather.json

```
{
  "other": {
    "cod": 200,
    "calctime": 0.006,
    "cnt": 110
  },
  "cities": [
  ...
  ]
}
```

然后用以下方式启动 `json-server`：

```
json-server -p 5000 ./cities-weather.json
```

在浏览器中访问 http://localhost:5000/cities，应该可以看到如图 6-1 所示的结果。



图 6-1：在浏览器中访问 `json-server` 提供的开放天气数据

将需要测试的 JSON 数据部署为模拟 API 后，接下来本章将对其进行单元测试。

6.4　设置单元测试环境

与之前的章节一样，本章中的所有单元测试均会在 Node.js 环境中通过 Mocha/Chai 来进行。在继续阅读前，需要确保已成功设置测试环境。如尚未安装 Node.js，可参考 A.2 节和 A.2.5 节中的内容。如需依照本节的描述运行代码示例中的项目，可使用 `cd` 命令切换到 chapter-6/cities-weather-test 目录，并执行以下命令来安装项目依赖：

```
npm install
```

如需手动创建本节中的 Node.js 项目，可参考本书在 GitHub 上的相关指导步骤。

设置好测试环境后，就可以开始使用 JSONPath 和其他的 JSON 搜索类库了。

6.5 比较JSON搜索类库和工具

了解了 JSON 搜索的一些基本知识后，本节将比较以下类库和工具：

- JSONPath
- JSON Pointer
- `jq`

6.5.1 JSONPath

JSONPath 是 Stefan Goessner 于 2007 年开发的，可用于对 JSON 文档进行搜索和数据抽提操作。该类库最开始是用 JavaScript 开发的，但由于其广为流行，如今大多数现代编程语言 / 平台都对其有着良好的支持。

1. JSONPath查询语法

JSONPath 的查询语法基于 XPath（一种用于搜索 XML 文档的查询语言）。表 6-1 列举了以城市天气为例的一些 JSONPath 查询语句。

表6-1：JSONPath查询语句

JSONPath查询语句	描述
`$.cities`	获取 `cities` 数组中的所有元素
`$.cities.length`	获取 `cities` 数组中的元素数目
`$.cities[0::2]`	以每隔一个元素进行抽取的方式从 `cities` 数组中获取数据。具体参考下述有关 `slice()` 的描述
`$.cities[(@.length-1)]` 或 `$.cities[-1:]`	获取 `cities` 数组中的最后一个元素
`$..weather`	获取所有的 `weather` 子元素
`$.cities[:3]`	获取 `cities` 数组的前三个元素
`$.cities[:3].name`	获取 `cities` 数组前三个元素的城市名
`$.cities[?(@.main.temp > 84)]`	获取 `temp` 值大于 84 的所有城市
`$.cities[?(@.main.temp >= 84 && @.main.temp <= 85.5)]`	获取 `temp` 值在 84~85.5 的所有城市
`$.cities[?(@.weather[0].main == 'Clouds')]`	获取天气类型为"多云"的所有城市
`$.cities[?(@.weather[0].main.match(/Clo/))]`	使用正则表达式来获取天气类型为"多云"的所有城市

以上查询语句示例使用了 JSONPath 的一些关键词和关键标识符。

- `$` 表示 JSON 文档的根节点对象。
- `..` 表示拥有特定名称的所有元素和子元素。
- `[]` 用于表示数组查询语句，其索引值则基于 JavaScript 中的 `slice()` 函数。对于 `slice()` 函数，Mozilla 开发者网络（Mozilla Developer Network，MDN）提供了非常全面的介绍。以下是对 JSONPath 中 `slice()` 的简单概览。
  - 提供了选择数组中部分内容的功能。

- – 与 JavaScript 中的 `slice()` 一样，`begin` 参数表示起始索引值（第一个值为 0），如忽略则其默认值为 0。
  - – 与 JavaScript 中的 `slice()` 一样，`end` 参数表示结束索引值（索引值刚好为 `end` 参数值的元素不包含在内），如忽略则默认为数组长度值。
  - – `step` 参数是 JSONPath 中新增的，用于表示元素遍历抽取时的步长，其默认值为 1。当 `step` 值为 1 时，返回的是数组从 `begin` 到 `end` 区域内的所有元素；当 `step` 值为 2 时，返回的是数组从 `begin` 到 `end` 区域内每隔一个元素抽取出的元素集合。以此类推。
- `@` 表示当前元素。
- `[?(...)]` 用于执行条件搜索。小括号内可以放置任意的 JavaScript 表达式，其中包括条件判断表达式（如 == 或 >）和正则表达式。

2. JSONPath在线测试工具

在没有编写任何代码的情况下，可以使用一些在线的 JSONPath 测试工具来实践 JSONPath 查询语句。在这些工具中，我最喜欢 Kazuki Hamasaki 提供的测试器。只需将 data/cities-weather.json 文档（参见本章代码示例）中的内容粘贴到左侧的输入框中，然后再输入 JSONPath 查询语句，即可在页面右侧区域观察到搜索结果。如图 6-2 所示。

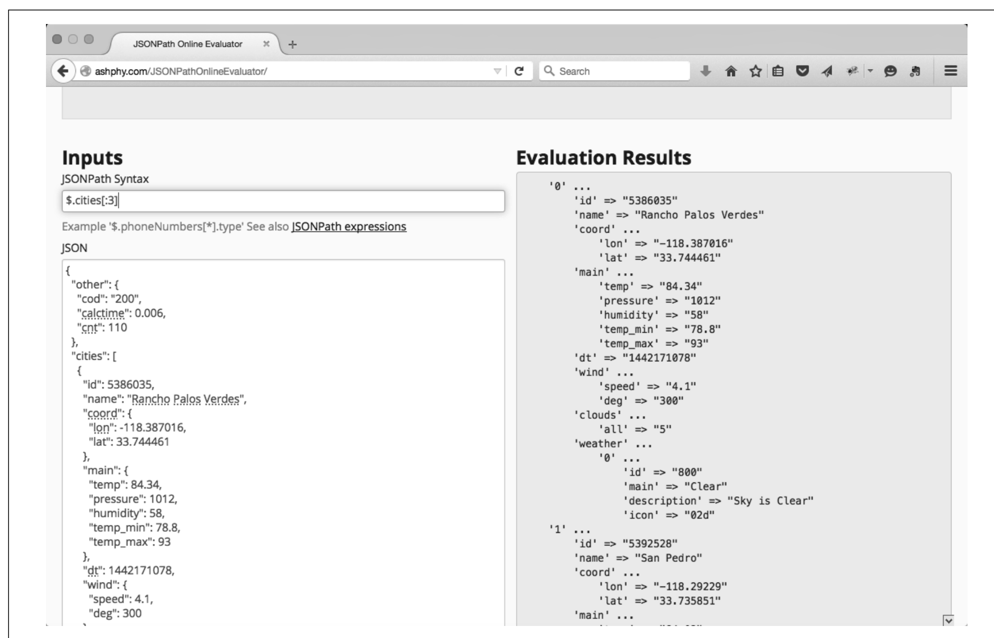


图 6-2：用 JSONPath 在线工具搜索开放天气 API 数据

3. JSONPath单元测试

例 6-2 中的单元测试练习了之前表格中的一些示例 JSONPath 查询语句。该代码使用 `jsonpath` 这一 Node.js 模块，对由本机运行的城市 API 所返回的 JSON 数据进行搜索。有关 `jsonpath` 模块的详细信息，可参考其 GitHub 主页。

例 6-2　cities-weather-test/test/jsonpath-spec.js

```
'use strict';

/* 声明：城市天气数据由OpenWeatherMap API通过
   Creative Commons Share A Like许可证发布。
   为兼容json-server，数据经过了一定的修改。
   此操作并不意味着得到了许可方的背书。

   此代码通过Creative Commons Share A Like许可证发布。
*/

var expect = require('chai').expect;
var jp = require('jsonpath');
var unirest = require('unirest');

describe('cities-jsonpath', function() {
  var req;

  beforeEach(function() {
    req = unirest.get('http://localhost:5000/cities')
      .header('Accept', 'application/json');
  });

  it('should return a 200 response', function(done) {
    req.end(function(res) {
      expect(res.statusCode).to.eql(200);
      expect(res.headers['content-type']).to.eql(
        'application/json; charset=utf-8');
      done();
    });
  });

  it('should return all cities', function(done) {
    req.end(function(res) {
      var cities = res.body;

      expect(cities.length).to.eql(110);
      done();
    });
  });

  it('should return every other city', function(done) {
    req.end(function(res) {
      var cities = res.body;
      var citiesEveryOther = jp.query(cities, '$[0::2]');

      expect(citiesEveryOther[1].name).to.eql('Rosarito');
      expect(citiesEveryOther.length).to.eql(55);
      done();
    });
  });

  it('should return the last city', function(done) {
    req.end(function(res) {
```

```
    var cities = res.body;
    var lastCity = jp.query(cities, '$[(@.length-1)]');

    expect(lastCity[0].name).to.eql('Moreno Valley');
    done();
  });
});

it('should return the 1st 3 cities', function(done) {
  req.end(function(res) {
    var cities = res.body;
    var citiesFirstThree = jp.query(cities, '$[:3]');
    var citiesFirstThreeNames = jp.query(cities, '$[:3].name');

    expect(citiesFirstThree.length).to.eql(3);
    expect(citiesFirstThreeNames.length).to.eql(3);
    expect(citiesFirstThreeNames).to.eql(['Rancho Palos Verdes',
      'San Pedro', 'Rosarito'
    ]);

    done();
  });
});

it('should return cities within a temperature range', function(done) {
  req.end(function(res) {
    var cities = res.body;
    var citiesTempRange = jp.query(cities,
      '$[?(@.main.temp >= 84 && @.main.temp <= 85.5)]'
    );

    for (var i = 0; i < citiesTempRange.length; i++) {
      expect(citiesTempRange[i].main.temp).to.be.at.least(84);
      expect(citiesTempRange[i].main.temp).to.be.at.most(85.5);
    }

    done();
  });
});

it('should return cities with cloudy weather', function(done) {
  req.end(function(res) {
    var cities = res.body;
    var citiesWeatherCloudy = jp.query(cities,
      '$[?(@.weather[0].main == "Clouds")]'
    );

    checkCitiesWeather(citiesWeatherCloudy);
    done();
  });
});

it('should return cities with cloudy weather using regex', function(done) {
  req.end(function(res) {
    var cities = res.body;
```

```
      var citiesWeatherCloudyRegex = jp.query(cities,
        '$[?(@.weather[0].main.match(/Clo/))]'
      );

      checkCitiesWeather(citiesWeatherCloudyRegex);
      done();
    });
  });

  function checkCitiesWeather(cities) {
    for (var i = 0; i < cities.length; i++) {
      expect(cities[i].weather[0].main).to.eql('Clouds');
    }
  }
});
```

对于以上示例，需要注意以下几点。

- 测试代码在 Mocha 的 `beforeEach()` 方法中配置 `unirest` 的 URI 和 `Accept` 头部，以避免配置代码的重复。在 `describe` 语句所定义的范围内，Mocha 会在执行每个测试用例（即 `it` 语句）前先运行一遍 `beforeEach()` 方法。
- 测试用例使用 `expect` 断言风格，并练习单个或更多 JSONPath 查询语句。
- 调用 `jsonpath` 模块的工作机制如下。
  - `jp.query()` 语句接受 JavaScript 对象作为第一个参数，同时接受 JSONPath 查询语句（以字符串表示）作为第二个参数，然后以 JavaScript 对象的形式同步地返回结果。
- 因为向 `json-server` 发送的请求中包含了 `cities` 这一数组的字段名（URI 中包含 `cities`），所以上述示例中所有的 JSONPath 查询语句都忽略了开头的 `.cities` 部分。
  - URI 地址是 http://localhost:5000/cities。
  - 使用 `$[:3]`（而不是 `$.cities[:3]`）来获取前 3 个城市的数据。

可以新开命令行终端并执行以下命令来运行上面的单元测试程序：

```
cd cities-weather-test

npm test
```

运行后可观察到以下结果：

```
json-at-work => npm test

...

> mocha test

...

cities-jsonpath
  ✓should return a 200 response
  ✓should return all cities
  ✓should return every other city
  ✓should return the last city
  ✓should return 1st 3 cities
```

```
    ✓should return cities within a temperature range
    ✓should return cities with cloudy weather
    ✓should return cities with cloudy weather using regex

...
```

如果在以上示例的任意测试代码中调用 `console.log()` 来打印相关变量，那么可以看到 `jsonpath` 模块将返回以名称－值对表示的合法 JSON 文档。

4. 在其他平台中使用JSONPath

JSONPath 的使用并不局限于 JavaScript 和 Node.js 环境。事实上，大多数主流平台都对 JSONPath 有着很不错的支持。这些平台包括：

- Ruby on Rails
- Python
- Java

除此之外，还存在很多其他的优秀 JSONPath 类库，但使用前请确认其遵循了 JSONPath 的语法（参考 Stefan Goessner 的相关文章），否则就谈不上是严格意义的 JSONPath。借用《公主新娘》中的一句话来说："你一直在使用那个词，但我觉得它并不是你以为的那个意思。"

5. JSONPath记分结果

根据本章开头所列举的评估标准，表 6-2 显示了 JSONPath 的记分结果。

表6-2：JSONPath记分结果

关注度	Y
开发社区	Y
平台	JavaScript、Node.js、Java、Ruby on Rails
易于入门	Y
标准	N

JSONPath 提供了丰富的搜索特性，并且能在大多数主流平台上工作。其唯一缺陷在于 JSONPath 并不是一项标准，同时也缺乏命令行界面的相关实现。不过，这对于实际使用来说无伤大雅。JSONPath 广为接受且有着较高的社区使用率，同时也提供了优秀的在线测试工具。对于访问、搜索 JSON 文档并获取数据子集来说，JSONPath 可以有效减少代码量。

6.5.2 JSON Pointer

JSON Pointer 是用于获取 JSON 文档中某个特定值的一项标准。关于其细节，可参考 JSON Pointer 的标准文档。设计 JSON Pointer 的主要目的在于支持 JSON Schema 标准中的 `$ref` 功能（在同一 Schema 中对校验规则的定位引用，详见第 5 章）。

1. JSON Pointer查询语法

思考以下文档：

```
{
  "cities": [
    {
      "id": 5386035,
      "name": "Rancho Palos Verdes"
    },
    {
      "id": 5392528,
      "name": "San Pedro"
    },
    {
      "id": 5358705,
      "name": "Huntington Beach"
    }
  ]
}
```

表 6-3 描述了这一文档中常用的 JSON Pointer 查询语法。

表6-3：JSON Pointer查询

JSON Pointer查询	描述
`/cities`	获取数组中所有的城市
`/cities/0`	获取第一个城市
`/cities/1/name`	获取第二个城市的名称

JSON Pointer 的查询语法非常简单，其工作机制如下。

- / 表示路径分隔符。
- 数组的索引值以 0 开头。

可以注意到，在 JSON Pointer 标准中，当发起查询操作时，返回的结果只会包含数据值，而不会包含相关的键名。

2. JSON Pointer单元测试

例 6-3 中的单元测试实践了之前表格中的一些示例 JSON Pointer 查询语句。该代码使用 json-pointer 这一 Node.js 模块对 `cities` API 所返回的 JSON 数据进行搜索。有关 json-pointer 模块的详细信息，可参考其 GitHub 主页。

例 6-3　cities-weather-test/test/json-pointer-spec.js

```
'use strict';
/* 声明：城市天气数据由OpenWeatherMap API通过
   Creative Commons Share A Like许可证发布。
   为兼容json-server，数据经过了一定的修改。
   此操作并不意味着得到了许可方的背书。

   此代码通过Creative Commons Share A Like许可证发布。
*/

var expect = require('chai').expect;
var pointer = require('json-pointer');
var unirest = require('unirest');
```

```
describe('cities-json-pointer', function() {
  var req;

  beforeEach(function() {
    req = unirest.get('http://localhost:5000/cities')
                 .header('Accept', 'application/json');
  });

  it('should return a 200 response', function(done) {
    req.end(function(res) {
      expect(res.statusCode).to.eql(200);
      expect(res.headers['content-type']).to.eql(
                 'application/json; charset=utf-8');
      done();
    });
  });

  it('should return the 1st city', function(done) {
    req.end(function(res) {
      var cities = res.body;
      var firstCity = null;

      firstCity = pointer.get(cities, '/0');
      expect(firstCity.name).to.eql('Rancho Palos Verdes');
      expect(firstCity.weather[0].main).to.eql('Clear');
      done();
    });
  });

  it('should return the name of the 2nd city', function(done) {
    req.end(function(res) {
      var cities = res.body;
      var secondCityName = null;

      secondCityName = pointer.get(cities, '/1/name');
      expect(secondCityName).to.eql("San Pedro");
      done();
    });
  });
});
```

对于以上示例，需要注意以下几点。

- 测试用例使用 `expect` 断言风格来运行示例 JSON Pointer 查询语句。
- 调用 `json-pointer` 模块的工作机制如下。
  - `pointer.get()` 语句接受 JavaScript 对象作为第一个参数，同时接受 JSON Pointer 查询语句（以字符串表示）作为第二个参数，然后以 JavaScript 对象的形式同步地返回结果。
- 因为向 `json-server` 发送的请求中包含了 `cities` 这一数组的字段名（URI 中包含 `cities`），所以上述示例中的所有 JSON Pointer 查询语句都忽略了开头的 `.cities` 部分。
  - URI 地址是 http://localhost:5000/cities。
  - 使用 `/0`（而不是 `/cities/0`）来获取第一个城市的数据。

可以新开命令行终端并执行以下命令来运行上面的单元测试程序：

```
cd cities-weather-test

npm test
```

运行后可观察到以下结果：

```
json-at-work => npm test

...

> mocha test

...

cities-json-pointer
  ✓should return a 200 response
  ✓should return the 1st city
  ✓should return the name of the 2nd city

...
```

在以上示例的 `should return the 1st city` 测试用例中，如果调用 `console.log()` 来打印 `firstCity` 变量，那么可以看到 `json-pointer` 模块将返回以名称 – 值对表示的合法 JSON 文档。

3. 在其他平台中使用JSON Pointer

除了 Node.js，大多数主流平台都包含 JSON Pointer 相关类库：

- Ruby on Rails；
- Python；
- Java，Jackson 现在支持 JSON Pointer 的查询语法。作为 JSR 374 标准的一部分（JSON 处理 1.1 的 Java API），JavaEE 8 将对 JSON Pointer 提供原生支持。

有些工具号称实现了 JSON Pointer，但事实上却并未遵循 JSON Pointer 标准。因此，当评估 JSON Pointer 类库或工具时，请确保其遵循了 RFC 6901 标准。再次强调，如果某个工具没有明确提到 RFC 6901，那么该工具所做的就不是 JSON Pointer 处理。

4. JSON Pointer记分结果

根据评估标准，表 6-4 显示了 JSON Pointer 的记分结果。

表6-4：JSON Pointer记分结果

关注度	Y
开发社区	Y
平台	JavaScript、Node.js、Java、Ruby on Rails
易于入门	Y
标准	Y — RFC 6901

JSON Pointer 提供了有限的搜索功能。每次查询只返回 JSON 文档中某一个字段的值。设计 JSON Pointer 的主要目的是支持 JSON Schema 中的 `$ref` 语法。

6.5.3 jq

jq 是一个提供了命令行界面的 JSON 搜索工具，其功能包括过滤 JSON 和截取数组。根据其 GitHub 主页，jq 有点像是 JSON 中的 sed。不过，对 jq 的使用并不仅限于命令行；通过引入一些不错的类库，你可以在自己喜欢的单元测试框架中使用 jq（详见本节中的“jq 单元测试”部分）。

1. 整合cURL

对于 Web API，UNIX 社区中的很多人会在命令行中使用 cURL 来发起 HTTP 调用。除了 HTTP，cURL 还提供了多种协议的数据通信功能。如需安装 cURL，请参考 A.7 节中的内容。

首先，我们在命令行中使用 cURL 向城市 API 发起 GET 请求，如下所示：

```
curl  -X GET 'http://localhost:5000/cities'
```

从城市 API 获取 JSON 响应后，我们通过管道将响应内容输出到 jq，让其对数据进行过滤操作。以下是一个简单示例：

```
curl  -X GET 'http://localhost:5000/cities' | jq .[0]
```

运行该命令后可以看到以下结果：

```
json-at-work => curl  -X GET 'http://localhost:5000/cities' | jq .[0]
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 57510  100 57510    0     0   806k      0 --:--:-- --:--:-- --:--:--  802k
{
  "id": 5386035,
  "name": "Rancho Palos Verdes",
  "coord": {
    "lon": -118.387016,
    "lat": 33.744461
  },
  "main": {
    "temp": 84.34,
    "pressure": 1012,
    "humidity": 58,
    "temp_min": 78.8,
    "temp_max": 93
  },
  "dt": 1442171078,
  "wind": {
    "speed": 4.1,
    "deg": 300
  },
  "clouds": {
    "all": 5
  },
  "weather": [
    {
      "id": 800,
      "main": "Clear",
      "description": "Sky is Clear",
      "icon": "02d"
    }
  ]
}
```

对于以上示例，需要注意以下两点。

- cURL 向 OpenWeatherMap API 发起 HTTP GET 请求，并将 JSON 响应导入标准输出。
- jq 从标准输入中读取 JSON，从这一 API 数据中选择第一个城市，并将第一个城市的 JSON 结果显示到标准输出。

在 API 开发工具中，cURL 是非常有价值且强大的。cURL 支持 API 测试中所需的所有 HTTP 方法（GET、POST、PUT 和 DELETE）。我们只是介绍了 cURL 的一些皮毛，如需学习更多有关 cURL 的知识，可参考其官方网站。

2. jq查询语法

表 6-5 展示了一些基本的 jq 查询语句。

表6-5：jq查询语句

jq查询语句	描述
`.cities[0]`	获取第一个城市。jq 的数组过滤索引值以 0 开头
`.cities[-1]`	获取最后一个城市。索引值 -1 表示数组中的最后一个元素
`.cities[0:3]`	获取前三个城市，其中 0 为起始索引（自身包含在内），3 为结尾索引（自身不包含在内）
`.cities[:3]`	获取前三个城市。这一语句忽略了起始索引，是一种快捷写法
`.cities[] | select (.main.temp >= 80 and (.main.temp_min >= 79and .main.temp_max <= 92))`	获取满足以下要求的所有城市：当前温度 ≥ 80 华氏度，且温度范围在 79~92 华氏度（包括 79 度和 92 度）

以下命令展示了如何在命令行中使用 jq 查询来获取最后一个城市：

```
cd chapter-6/data

jq '.cities[-1]' cities-weather.json
```

运行命令后可以看到以下结果：

```
json-at-work => jq '.cities[-1]' cities-weather.json
{
  "id": 5374732,
  "name": "Moreno Valley",
  "coord": {
    "lon": -117.230591,
    "lat": 33.937519
  },
  "main": {
    "temp": 87.84,
    "pressure": 1013,
    "humidity": 42,
    "temp_min": 82.4,
    "temp_max": 98.6
  },
  "dt": 1442171075,
  "wind": {
    "speed": 1,
    "deg": 0
  },
  "clouds": {
    "all": 1
  },
  "weather": [
    {
      "id": 800,
      "main": "Clear",
      "description": "Sky is Clear",
      "icon": "01d"
    }
  ]
}
```

接下来，我们将介绍一个更具体的示例。

3. jq在线测试工具——jqPlay

jqPlay 是一个基于 Web 的 jq 测试器，提供了对 JSON 数据进行 jq 查询的功能。如需测试 jqPlay，可以执行以下步骤来获取由前三个城市的 id、name 对象所组成的数组。

(1) 访问 jqPlay 网站，将 chapter-6/data/cities-weather.json 文件的内容粘贴到网页左侧的 JSON 文本框区域中。

(2) 将以下 jq 查询语句粘贴到 Filter 文本输入框中：[.cities[0:3] | .[] | { id, name }]。

可以看到如图 6-3 所示的结果。

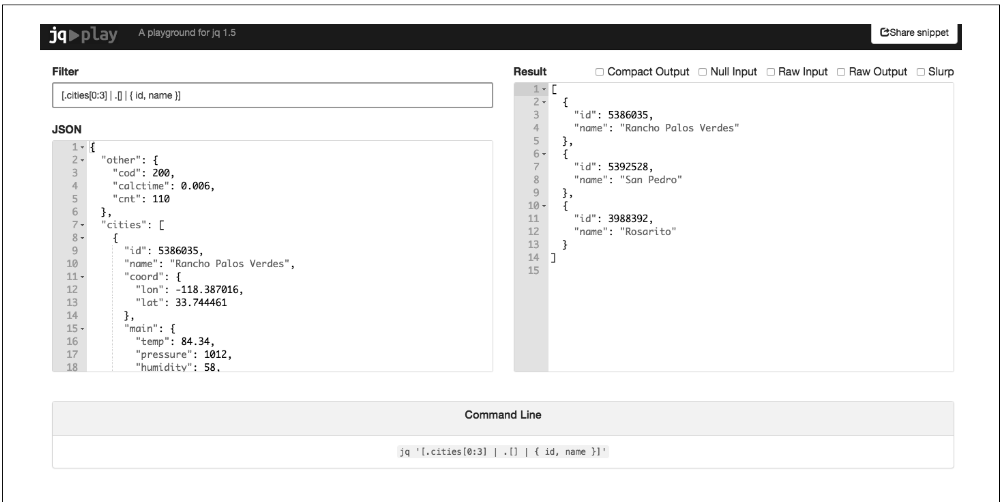


图 6-3：用 jqPlay 搜索开放天气 API 数据

以下是对 [.cities[0:3] | .[] | { id, name }] 查询语句的详细解释。

- | 符号将多个过滤操作串起来（链式级联）。
- .cities[0:3] 从城市数组中选择前三个元素所组成的子数组。
- .[] 返回该子数组中的所有元素。
- { id, name } 仅选择元素中的 id 和 name 字段。
  - 大括号（{ 和 }）表示创建一个新的对象。
  - id 和 name 表示该新对象只包含这两个字段。
- 最外围的中括号（[和]）将结果转换为数组。

浏览 jqPlay 网页的底部，你可以看到一张速查表，其中包含了更多示例及文档的链接，如图 6-4 所示。

Cheatsheet

Click on the icons (▤) in the table below to see examples.

`.`	unchanged input	`,`	feed input into multiple filters
`.foo, .foo.bar, .foo?`	value at key	`|`	pipe output of one filter to the next filter
`.[], .[]?, .[2], .[10:15]`	array operation	`select(foo)`	input unchanged if foo returns true
`[], {}`	array/object construction	`map(foo)`	invoke filter foo for each input
`length`	length of a value	`if-then-else-end`	conditionals
`keys`	keys in an array	`\(foo)`	string interpolation

View source on GitHub

图 6-4：jqPlay 中的 jq 速查表

4. jq-tutorial

除了在线测试器，Node.js 社区还提供了以 npm 模块形式发布的一个很不错的 jq 教程。可以用以下方式安装该教程：

```
npm install -g jq-tutorial
```

在命令行中运行 jq-tutorial，可以看到以下结果：

```
json-at-work => jq-tutorial
Run jq-tutorial with one of the following:
  * pick
  * objects
  * mapping
  * filtering
  * output
  * reduce
```

这一命令能够显示所有的 jq 教程。可以选择其中一个教程：

```
jq-tutorial objects
```

该教程展示了如何在 jq 中使用对象。通过阅读所有的学习条目，你可以有效增强自己在 jq 上的技能水平。

5. jq单元测试

例 6-4 中的单元测试实践了之前的一些示例 jq 查询语句。该代码使用 node-jq 这一 Node.js 模块，对由本机运行的 cities API 所返回的 JSON 数据进行了搜索。有关 node-jq 模块的详细信息，可以参考其 GitHub 主页。

例 6-4　cities-weather-test/test/jq-spec.js

```
'use strict';
/* 声明：城市天气数据由OpenWeatherMap API通过
   Creative Commons Share A Like许可证发布。
   为兼容json-server，数据经过了一定的修改。
   此操作并不意味着得到了许可方的背书。

   此代码通过Creative Commons Share A Like许可证发布。
```

```
*/

var expect = require('chai').expect;
var jq = require('node-jq');
var unirest = require('unirest');
var _ = require('underscore');

describe('cities-jq', function() {
  var req;

  beforeEach(function() {
    req = unirest.get('http://localhost:5000/cities')
      .header('Accept', 'application/json');
  });

  it('should return a 200 response', function(done) {
    req.end(function(res) {
      expect(res.statusCode).to.eql(200);
      expect(res.headers['content-type']).to.eql(
        'application/json; charset=utf-8');
      done();
    });
  });

  it('should return all cities', function(done) {
    req.end(function(res) {
      var cities = res.body;

      expect(cities.length).to.eql(110);
      done();
    });
  });

  it('should return the last city', function(done) {
    req.end(function(res) {
      var cities = res.body;

      jq.run('.[-1]', cities, {
          input: 'json'
        })
        .then(function(lastCityJson) { // 返回JSON字符串。
          var lastCity = JSON.parse(lastCityJson);
          expect(lastCity.name).to.eql('Moreno Valley');
          done();
        })
        .catch(function(error) {
          console.error(error);
          done(error);
        });
    });
```

```
});

it('should return the 1st 3 cities', function(done) {
  req.end(function(res) {
    var cities = res.body;

    jq.run('.[:3]', cities, {
        input: 'json'
      })
      .then(function(citiesFirstThreeJson) { // 返回JSON字符串。
        var citiesFirstThree = JSON.parse(citiesFirstThreeJson);
        var citiesFirstThreeNames = getCityNamesOnly(
          citiesFirstThree);

        expect(citiesFirstThree.length).to.eql(3);
        expect(citiesFirstThreeNames.length).to.eql(3);
        expect(citiesFirstThreeNames).to.eql([
          'Rancho Palos Verdes',
          'San Pedro', 'Rosarito'
        ]);

        done();
      })
      .catch(function(error) {
        console.error(error);
        done(error);
      });
  });
});

function getCityNamesOnly(cities) {
  return _.map(cities,
    function(city) {
      return city.name;
    });
}

it('should return cities within a temperature range', function(done) {
  req.end(function(res) {
    var cities = res.body;

    jq.run(
        '[.[] | select (.main.temp >= 84 and .main.temp <= 85.5)]',
        cities, {
          input: 'json'
        })
      .then(function(citiesTempRangeJson) { // 返回JSON字符串。
        var citiesTempRange = JSON.parse(citiesTempRangeJson);

        for (var i = 0; i < citiesTempRange.length; i++) {
```

```
          expect(citiesTempRange[i].main.temp).to.be.at.least(84);
          expect(citiesTempRange[i].main.temp).to.be.at.most(85.5);
        }

        done();
      })
      .catch(function(error) {
        console.error(error);
        done(error);
      });
  });
});

it('should return cities with cloudy weather', function(done) {
  req.end(function(res) {
    var cities = res.body;

    jq.run(
        '[.[] | select(.weather[0].main == "Clouds")]',
        cities, {
          input: 'json'
        })
      .then(function(citiesWeatherCloudyJson) { // 返回JSON字符串。
        var citiesWeatherCloudy = JSON.parse(
          citiesWeatherCloudyJson);

        checkCitiesWeather(citiesWeatherCloudy);

        done();
      })
      .catch(function(error) {
        console.error(error);
        done(error);
      });
  });
});

it('should return cities with cloudy weather using regex', function(done) {
  req.end(function(res) {
    var cities = res.body;

    jq.run(
        '[.[] | select(.weather[0].main | test("^Clo"; "i"))]',
        cities, {
          input: 'json'
        })
      .then(function(citiesWeatherCloudyJson) { // 返回JSON字符串。
        var citiesWeatherCloudy = JSON.parse(
          citiesWeatherCloudyJson);
```

```
          checkCitiesWeather(citiesWeatherCloudy);

          done();
        })
        .catch(function(error) {
          console.error(error);
          done(error);
        });
    });
  });

  function checkCitiesWeather(cities) {
    for (var i = 0; i < cities.length; i++) {
      expect(cities[i].weather[0].main).to.eql('Clouds');
    }
  }

});
```

对于以上示例，需要注意以下几点。

- 测试代码在 Mocha 的 `beforeEach()` 方法中配置 `unirest` 的 URI 和 `Accept` 头部，从而避免配置代码的重复。在 `describe` 语句所定义的范围内，Mocha 会在执行每个测试用例（即 `it` 语句）前先运行一遍 `beforeEach()` 方法。
- 测试用例使用 `expect` 断言风格来练习单个或更多 `jq` 查询语句。
- 调用 `node-jq` 模块的工作机制如下。`jq.run()` 运行时会进行以下操作。
  - 接受 `jq` 查询语句（以字符串表示）作为第一个参数。
  - 第二个对象参数是可选的，表明了输入的类型。
    - ♦ `{ input: 'JSON' }` 表示一个 JavaScript 对象。当 `unirest` 向 `json-server` 提供的模拟 API 发起 HTTP 调用时，返回的结果为对象，因此上述单元测试使用了这一选项。
    - ♦ `{ input: 'file' }` 表示一个 JSON 文件。该选项为默认选项，如果调用 `jq.run()` 时不指定输入类型，则输入的即为 JSON 文件。
    - ♦ `{ input: 'string' }` 表示一个 JSON 字符串。
  - 使用 ES6 中的 JavaScript Promise 来异步返回 JSON 字符串结果。在本例中，单元测试需要执行的相关操作如下。
    - ♦ 将测试代码包含在 Promise 的 `then` 和 `catch` 结构中。
    - ♦ 使用 `JSON.parse()` 将结果解析为对应的 JavaScript 对象结构。
  - 如需学习更多有关 Promise 的新语法，可访问 MDN 网站。
- 因为向 `json-server` 发送的请求中包含了 `cities` 这一数组的字段名（URI 中包含 `cities`），所以上述示例中的所有 `jq` 查询语句都忽略了开头的 `.cities` 部分。
  - URI 地址是 http://localhost:5000/cities。
  - 使用 `$[:3]`（而不是 `$.cities[:3]`）来获取前三个城市的数据。

可以新开命令行终端并执行以下命令来运行上面的单元测试程序：

```
cd cities-weather-test

npm test
```

运行后可观察到以下结果：

```
json-at-work => npm test

...

> mocha test

...

  cities-jq
    ✓should return a 200 response
    ✓should return all cities
    ✓should return the last city
    ✓should return the 1st 3 cities
    ✓should return cities within a temperature range
    ✓should return cities with cloudy weather
    ✓should return cities with cloudy weather using regex

...
```

如果在以上示例的任意测试代码中调用 `console.log()` 来打印相关变量，可以看到 `node-jq` 模块将返回以名称 – 值对表示的合法 JSON 文档。

6. 在其他平台中使用jq

除了 Node.js，其他主流平台也都包含了 `jq` 类库。

Ruby

可以在 RubyGems.org 网站上找到 `ruby-jq` gem，也可以参考其 GitHub 主页。

Java

Java 中的 Jackson 类库（详见第 4 章）包含了 `jackson-jq` 插件。

7. jq记分结果

根据评估标准，表 6-6 显示了 `jq` 的记分结果。

表6-6：jq记分结果

关注度	Y
开发社区	Y
平台	CLI—Linux/macOS/Windows、Node.js、Java、Ruby on Rails
易于入门	Y
标准	N

`jq` 是一个优秀的工具，具体原因如下。

- 大多数编程语言都对 `jq` 有着良好的支持。

- 文档质量较高。
- 提供丰富的搜索和过滤功能。
- 可以通过管道将查询结果输出到 UNIX 命令行界面工具（如 `sort`、`grep` 和 `uniq` 等）进行处理。
- 可以在命令行中很好地与 cURL 这一 HTTP 客户端工具进行协同工作。
- 具有非常优秀的在线测试工具。`jqPlay` 使得你能够在一个简单的 Web 界面中测试 `jq` 查询操作。该工具能够提供快速反馈，因此可在不编写任何代码的情况下，迭代得到最终的查询方案。
- 存在有用的交互式教程（详见本节的“`jq`-tutorial”部分）。

`jq` 的唯一缺陷在于其初始学习曲线较陡，其查询语法中的大量选项一开始可能会显得过于复杂，让人不知所措。不过，花在 `jq` 上的学习时间最终还是物有所值的。

至此，本章介绍了 `jq` 的一些基本知识。`jq` 提供了高质量的文档，在下列网站中也可找到更多详细信息：

- `jq` 手册；
- `jq` 教程；
- `jq` 实例；
- HyperPolyGlot JSON 工具（`jq`）；
- Ubuntu 中的 `jq` 手册页面。

6.6 搜索工具评估——总结概要

基于评估标准以及总体的可用性，我将 JSON 搜索类库按照以下顺序排名：

(1) `jq`
(2) JSONPath
(3) JSON Pointer

虽然 JSON Pointer 是一项标准，**可以**用于搜索 JSON 文档，但我还是将 JSONPath 置于 JSON Pointer 之上，原因如下。

- JSONPath 的查询语法更丰富。
- JSONPath 的查询语句可以返回文档中的多个元素。

不过出于以下原因，我最钟爱的 JSON 搜索工具还是 `jq`。

- 可以在命令行中工作（JSONPath 和 JSON Pointer 都不提供此功能）。如果需要在自动化运维环境中处理 JSON，则命令行工具是不可或缺的。
- 具有在线测试工具，这一点可以加速开发工作。
- 存在交互式教程。
- 提供丰富的查询语句。
- 大多数编程语言均提供良好的 `jq` 类库支持。
- 在 JSON 社区中的关注度较高。

我曾成功地使用 `jq` 来搜索其他 Web API（不是本章中的 OpenWeatherMap）提供的 JSON 响应；该响应包含超过两百万行的数据，而 `jq` 在生产环境中完美地执行了搜索操作。`jq` 在 JSON 社区中拥有很高的关注度，只要在网络上搜索“`jq` 教程”，就可以找到不少优秀的教程资源来帮助你更深入地学习。

6.7 本章回顾

本章介绍了一些广泛使用的 JSON 搜索类库和工具，并展示了如何测试搜索结果。至此，希望你已经接受了使用 JSON 搜索技术来减少工作量的理念，而不是继续手工编写相关工具代码。

6.8 内容预告

展示了如何高效搜索 JSON 文档后，第 7 章将介绍对 JSON 的转换操作。

第7章

JSON转换

有时应用程序会从多个 API 处获取数据，为了使得数据格式与应用架构相兼容，你需要转换 API 的 JSON 响应。

可以使用多种 JSON 转换技术在 JSON 文档和其他数据格式（如 HTML 或 XML）间进行转换，或者将 JSON 转换为新的结构。很多开发者已经比较熟悉其中一些类库（如 Mustache 和 Handlebars）；本章将以不落俗套的方式介绍这些工具的使用。除此之外，我们还将介绍 JSON-T 这样在技术社区不是那么有名，但在 JSON 社区却广为使用的类库。

7.1　JSON转换类型

典型的转换类型包括以下 3 种。

将 JSON 转换为 HTML

　　很多 Web 和移动应用程序都必须能够处理 API 返回的 JSON 数据，因此这一 JSON 转换类型是最常用的。

对 JSON 的格式进行转换

　　有时 Web API 返回的 JSON 响应并不是你所需要的，这时你就需要修改数据的格式以方便后续处理。在这种情况下，可以对值进行修改或者移除、添加字段来更改数据的结构。有些类库和 XML 中的 XSLT（eXtensible Stylesheet Language Transformations，可扩展样式表语言转换，用于转换 XML 文档）比较像，都通过使用单独的模板来描述转换规则。

将 JSON 转换为 XML

　　基于 SOAP/XML 的 Web Service 并未消亡，因此有时需要读取 XML 并将其转换成 JSON，从而与企业中基于 REST 和 JSON 的新应用程序保持兼容。相反，应用程序也可能需要向基于 SOAP/XML 的 Web Service 发送 XML 数据。在这种情况下，将 JSON 转换为 XML 是不可或缺的。

本章将介绍以下操作：

- 将 JSON 转换为 HTML；
- 将某个 JSON 文档转换为新的 JSON 结构；
- 在 XML 和 JSON 之间进行相互转换；
- 使用 JSON 转换类库；
- 编写单元测试来转换 Web API 返回的 JSON 文档。

7.2　选择JSON转换类库的标准

与 JSON 中的搜索一样，每种转换操作都有多个类库工具可用，因此作出选择还是比较困难的。本章将使用与第 6 章相同的评估标准。

关注度

　该工具是否广泛使用？当在网络上进行相关搜索时，能出来多少条搜索结果？

开发者社区

　代码是不是在 GitHub 上？维护的状态怎么样？

平台

　是否能跨平台工作？标准或类库接口是否得到了广泛支持？

易于入门

　文档是否齐全？安装难度如何？接口设计是否符合直觉？使用难度如何？需要编写多少代码？

标准

　是否存在相关标准（如 IETF、W3C 或 Ecma 等）？

7.3　测试输入数据

我们将使用与第 6 章相同的 OpenWeatherMap API 数据作为示例。原始的 OpenWeatherMap API 数据存放于 chapter-7/ data/cities-weather.json 文件中。出于简洁的目的，例 7-1 提供了该数据的缩减版。

例 7-1　data/cities-weather-short.json

```
{
  "cities": [
    {
      "id": 5386035,
      "name": "Rancho Palos Verdes",
      "coord": {
        "lon": -118.387016,
        "lat": 33.744461
      },
      "main": {
        "temp": 84.34,
        "pressure": 1012,
```

```
      "humidity": 58,
      "temp_min": 78.8,
      "temp_max": 93
    },
    "dt": 1442171078,
    "wind": {
      "speed": 4.1,
      "deg": 300
    },
    "clouds": {
      "all": 5
    },
    "weather": [
      {
        "id": 800,
        "main": "Clear",
        "description": "Sky is Clear",
        "icon": "02d"
      }
    ]
  },
  {
    "id": 5392528,
    "name": "San Pedro",
    "coord": {
    "lon": -118.29229,
    "lat": 33.735851
    },
    "main": {
      "temp": 84.02,
      "pressure": 1012,
      "humidity": 58,
      "temp_min": 78.8,
      "temp_max": 91
    },
    "dt": 1442171080,
    "wind": {
      "speed": 4.1,
      "deg": 300
    },
    "clouds": {
      "all": 5
    },
    "weather": [
      {
        "id": 800,
        "main": "Clear",
        "description": "Sky is Clear",
        "icon": "02d"
      }
    ]
  },
  {
    "id": 3988392,
    "name": "Rosarito",
```

```
"coord": {
  "lon": -117.033333,
  "lat": 32.333328
},
"main": {
  "temp": 82.47,
  "pressure": 1012,
  "humidity": 61,
  "temp_min": 78.8,
  "temp_max": 86
},
"dt": 1442170905,
"wind": {
  "speed": 4.6,
  "deg": 240
},
"clouds": {
  "all": 32
},
"weather": [
  {
    "id": 802,
    "main": "Clouds",
    "description": "scattered clouds",
    "icon": "03d"
  }
]
      }
    ]
  }
```

首先，我们来看一下将 JSON 转换为 HTML 的操作。

7.4　将JSON转换为HTML

大多数开发者应该已经很熟悉如何将 API 响应中的 JSON 转换为 HTML。对于这一转换，本节将介绍以下类库：

- Mustache
- Handlebars

7.4.1　目标HTML文档

我们将简化 7.3 节中的城市数据，并最终显示为如例 7-2 所示的 HTML 表格。

例 7-2　data/weather.html

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="UTF-8" />
    <title>OpenWeather - California Cities</title>
```

```
    <link rel="stylesheet" href="weather.css">
  </head>

  <body>
    <h1>OpenWeather - California Cities</h1>
    <table class="weatherTable">
      <thead>
        <tr>
          <th>City</th>
          <th>ID</th>
          <th>Current Temp</th>
        </tr>
      </thead>
      <tr>
        <td>Santa Rosa</td>
        <td>5201</td>
        <td>75</td>
      </tr>
    </table>
  </body>
</html>
```

在将示例 JSON 输入数据转换为目标 HTML 文档的过程中，我们将比较每个类库的相关操作。

7.4.2　Mustache

Mustache 使用声明式（无须编写逻辑代码）模板来转换数据格式。在本例中，我们会使用 Mustache 将 JSON 数据转换为 HTML 文档。因为使用的模板只包含一些简单的标签，没有 `if/then/else` 语句或循环结构，所以 Mustache 团队使用了**无逻辑**一词来描述其类库。根据说明文档，Mustache 可以解析模板文件中的标签，解析所用的值则来自应用程序的 `hash` 或对象。无论使用的是 Mustache 还是 Handlebars（Handlebars 中重新引入了一些条件逻辑），使用模板的优势在于：此方案从应用程序代码中抽取转换信息，并将其保存在外部文件中，从而实现了关注点分离。在不修改应用程序代码的情况下，外部模板使得你可以轻松地添加 / 删除数据格式，或者轻而易举地改变数据格式化方法。

如需了解更多信息，可参考以下网站：

- Mustache 官方网站；
- Mustache 的 GitHub 主页；
- Mustache 5 的说明文档。

1. Mustache模板语法

例 7-3 中的 Mustache 模板可将 OpenWeatherMap 的 JSON 数据转换为 HTML。

例 7-3　templates/transform-html.mustache

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="UTF-8" />
    <title>OpenWeather - California Cities</title>
```

```
    <link rel="stylesheet" href="weather.css">
  </head>
  <body>
    <h1>OpenWeather - California Cities</h1>
    <table class="weatherTable">
      <thead>
        <tr>
          <th>City</th>
          <th>ID</th>
          <th>Current Temp</th>
          <th>Low Temp</th>
          <th>High Temp</th>
          <th>Humidity</th>
          <th>Wind Speed</th>
          <th>Summary</th>
          <th>Description</th>
        </tr>
      </thead>
      {{#cities}}
        <tr>
          <td>{{name}}</td>
          <td>{{id}}</td>
          {{#main}}
            <td>{{temp}}</td>
            <td>{{temp_min}}</td>
            <td>{{temp_max}}</td>
            <td>{{humidity}}</td>
          {{/main}}
          <td>{{wind.speed}}</td>
          {{#weather.0}}
            <td>{{main}}</td>
            <td>{{description}}</td>
          {{/weather.0}}
        </tr>
      {{/cities}}
    </table>
  </body>

</html>
```

该模板的工作机制如下。

- 模板基于 HTML 文档，Mustache 则会使用 `cities` 数组中的数据来解析每个标签。
- 标签可表示单个字段，如 `{{temp}}`。
- 区块由起始标签（如 `{{#cities}}`）和结束标签（如 `{{/cities}}`）围起来。
  - 一个区块对应一个数组（如 `cities`）或一个对象（如 `main`）。
  - 一个区块为其内部的标签定义了上下文。如 `{{main}}` 区块内部的 `{{temp}}` 标签即可表示为 `{{main.temp}}`，对应 JSON 输入文档中的 main.temp 部分。
- 在标签中访问字段名时，可以使用数组索引。例如，`{{#weather.0}}` 表示输入 JSON 文档中的 `weather[0]` 部分。

接下来我们将在单元测试中使用城市数据来渲染模板。

2. Mustache单元测试

与之前的章节一样，本章中的所有单元测试均通过 Mocha/Chai 来运行。在继续阅读前，确保已成功设置测试环境。如尚未安装 Node.js，可参考 A.2 节和 A.2.5 节中的内容。如需依照本节的描述运行代码示例中的项目，可使用 `cd` 命令切换到 chapter-7/cities-weather-transform-test 目录，并执行以下命令来安装项目依赖：

```
npm install
```

如需手动创建本节中的 Node.js 项目，可参考本书在 GitHub 上的相关指导步骤。

例 7-4 中使用了以下 Node.js 模块。

Mustache
: 可以在 https://www.npmjs.com/package/mustache 中找到该模块，其 GitHub 主页为 https://github.com/janl/mustache.js。

`jsonfile`
: 我们将使用该模块从文件中读取 OpenWeatherMap 的 JSON 数据，并对数据进行解析。可以在 https://www.npmjs.com/package/jsonfile 中找到该模块，其 GitHub 主页为 https://github.com/jprichardson/node-jsonfile。

例 7-4 中的单元测试展示了实际应用中的 Mustache 转换操作。

例 7-4 cities-weather-transform-test/test/mustache-spec.js

```
'use strict';

/* 声明：城市天气数据由OpenWeatherMap API通过
   Creative Commons Share A Like许可证发布。
   为兼容json-server，数据经过了一定的修改。
   此操作并不意味着得到了许可方的背书。

   此代码通过Creative Commons Share A Like许可证发布。
*/

var expect = require('chai').expect;
var jsonfile = require('jsonfile');
var fs = require('fs');
var mustache = require('mustache');

describe('cities-mustache', function() {
  var jsonCitiesFileName = null;
  var htmlTemplateFileName = null;


  beforeEach(function() {
    var baseDir = __dirname + '/../..';

    jsonCitiesFileName = baseDir + '/data/cities-weather-short.json';
    htmlTemplateFileName = baseDir +
      '/templates/transform-html.mustache';
  });
```

```
  it('should transform cities JSON data to HTML', function(done) {
    jsonfile.readFile(jsonCitiesFileName, function(readJsonFileError,
      jsonObj) {
      if (!readJsonFileError) {
        fs.readFile(htmlTemplateFileName, 'utf8', function(
          readTemplateFileError, templateFileData) {
          if (!readTemplateFileError) {
            var template = templateFileData.toString();
            var html = mustache.render(template, jsonObj);

            console.log('\n\n\nHTML Output:\n' + html);
            done();
          } else {
            done(readTemplateFileError);
          }
        });
      } else {
        done(readJsonFileError);
      }
    });
  });
});
```

以上代码的工作机制如下。

- `beforeEach()` 方法会在每个单元测试用例执行前先运行一遍。在本例中，该方法构造了输入的 JSON 文件和 Mustache 模板的文件名。
- 在 `'should transform cities JSON data to HTML'` 这一单元测试用例中：
  - `jsonfile.readFile()` 读取输入的 JSON 文件，并将其解析为 JavaScript 对象（`jsonObj`）；
  - `fs.readFile()` 读取 Mustache 模板文件，并将其保存到一个 JavaScript 对象中；
  - 然后将 Mustache 模板转换为字符串；
  - `mustache.render()` 使用之前读取的 `jsonObj` 所提供的值将 Mustache 模板渲染为 HTML 文档。

运行单元测试前，打开终端并在命令行中以 5000 端口运行 `json-server`：

```
cd chapter-7/data

json-server -p 5000 ./cities-weather-short.json
```

然后新开命令行终端，执行以下命令来运行上面的单元测试：

```
cd chapter-7/cities-weather-transform-test

npm test
```

我们可以看到与目标 HTML 文档相似的 HTML 内容。

3. Mustache在线测试器

Architect 模板编辑器是一个非常优秀的在线测试工具，可以有效简化测试、开发 Mustache 模板的工作。当修改模板时，该工具可以实时显示渲染结果，因此是一个非常不错的选

择。像这样的所见即所得（WYSIWYG，What-You-See-Is-What-You-Get）工具可以有效加速开发与调试工作。

打开 Architect 在线工具，在 Engine 下拉菜单中选择 Mustache.js，然后将 Mustache 模板和输入的 JSON 粘贴到相应的输入框中。可以看到如图 7-1 所示的结果。

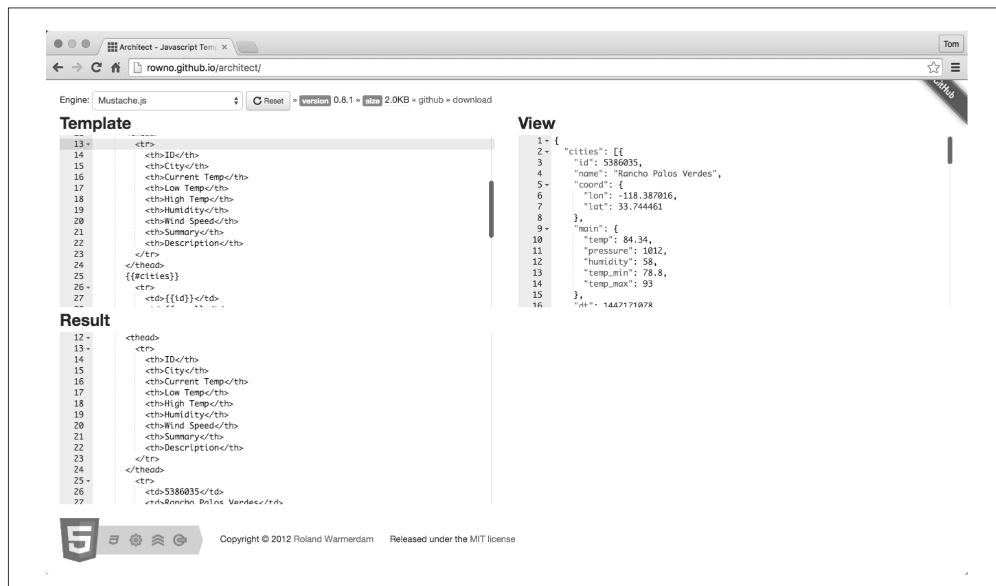


图 7-1：Architect：使用 Mustache 将 JSON 转换为 HTML

由于 Architect 还支持其他的一些模板引擎（如下一节将提到的 Handlebars），因此这一工具成为了我最喜欢使用的在线模板编辑器。

不过切记，这一 Web 应用程序是可以公开访问的。

- 粘贴到该应用程序的所有数据对于其他人来说都是可见的。因此，不要在此工具中使用敏感信息（个人隐私、财产信息等）。
- 数据量过大会影响浏览器的运行。我曾经最多使用过 10 000 行的 JSON 数据，但继续增加数据后应用程序就开始变得卡顿了。

4. 在命令行中使用Mustache

还可以直接在命令行中运行 Mustache。如果已经安装了 Node.js，则可在全局安装 Mustache 模块后，在本书示例代码路径下用命令行来运行该模块：

```
npm install -g mustache

cd chapter-7

mustache ./data/cities-weather-short.json \
  ./templates/transform-html.mustache > output.html
```

5. 在其他平台上使用Mustache

快速浏览 Mustache 网站就可以看到，Mustache 有着很好的跨平台支持。支持的平台包括：

- Node.js
- Ruby on Rails
- Java

6. Mustache记分结果

基于本章开头的评估标准，表 7-1 展示了 Mustache 的记分结果。

表7-1：Mustache记分结果

关注度	Y
开发社区	Y
平台	JavaScript、Node.js、Java、Ruby on Rails
易于入门	Y
标准	N

总而言之，Mustache 是一个强大、灵活且广为使用的模板引擎。虽然不是标准，但 Mustache 自身提供了非常可靠的说明文档。

接下来我们将介绍 Handlebars。

7.4.3 Handlebars

Handlebars 是 Mustache 的一个扩展，同样使用了 `hash` 或对象中的值来解析模板文件中的标签。Handlebars 与 Mustache 高度兼容，Mustache 的模板文件一般也能在 Handlebars 的引擎中正常工作。由于 HTML 转换操作非常简单，因此我们不会看到 Mustache 和 Handlebars 之间有什么区别。与 Mustache 相比，Handlebars 中添加了一些特性来增强转换操作，7.5 节将对此进行详细介绍。如需了解更多有关 Handlebars 的信息，可参考以下资源：

- Handlebars 官方网站（可点击 Learn More 按钮获取更多细节信息）；
- Handlebars 的 GitHub 主页。

1. Handlebars与Mustache间的差别

通过提供额外的功能，Handlebars 扩展了 Mustache。这些额外功能如下。

条件逻辑

　　Handlebars 提供了 `if` 和 `unless` 等内嵌的辅助语句。7.5 节将介绍如何使用 `unless`。

辅助语句

　　Handlebars 允许开发者通过注册自定义辅助语句的方式来扩展 Handlebars。每个自定义辅助语句都能提供一个额外的可用于模板的指令。比如，可以将 `speaker` 的 `firstName` 和 `lastName` 元素组合成新的 `{{fullName}}` 辅助语句。辅助语句非常强大，但本书不再对其进行深入介绍。如需了解更多有关 Handlebars 辅助语句的信息，可以参考其官网介绍页面，也可以阅读 Jasko Koyn 撰写的文章“Custom Helpers Handlebars.js Tutorial”。

Handlebars 的 GitHub 页面对 Handlebars 和 Mustache 之间的差别进行了全面说明。

2. Handlebars模板语法

我们使用例 7-5 中的 Handlebars 模板将输入的 JSON 转换为 HTML 文档。

例 7-5　templates/transform-html.hbs

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="UTF-8" />
    <title>OpenWeather - California Cities</title>
    <link rel="stylesheet" href="weather.css">
  </head>
  <body>
    <h1>OpenWeather - California Cities</h1>
    <table class="weatherTable">
      <thead>
        <tr>
          <th>ID</th>
          <th>City</th>
          <th>Current Temp</th>
          <th>Low Temp</th>
          <th>High Temp</th>
          <th>Humidity</th>
          <th>Wind Speed</th>
          <th>Summary</th>
          <th>Description</th>
        </tr>
      </thead>
      {{#each cities}}
      <tr>
        <td>{{id}}</td>
        <td>{{name}}</td>
        {{#main}}
          <td>{{temp}}</td>
          <td>{{temp_min}}</td>
          <td>{{temp_max}}</td>
          <td>{{humidity}}</td>
        {{/main}}
        <td>{{wind.speed}}</td>
        {{#each weather}}
          <td>{{main}}</td>
          <td>{{description}}</td>
        {{/each}}
      </tr>
      {{/each}}
    </table>
  </body>

</html>
```

以上模板的工作机制如下。

- Handlebars 会使用 `cities` 数组中的数据来解析每个标签。
- 标签可表示单个字段，如 `{{temp}}`。

- 区块由起始标签（如 {{#each cities}}）和结束标签（如 {{/cities}}）围起来。
  - 一个区块对应一个数组（如 cities）或一个对象（如 main）。
  - each 标签（如 {{#each cities}}）用于操作数组（本示例中为 cities）。
  - 一个区块为其内部的标签定义了上下文。如 {{main}} 区块内部的 {{temp}} 标签即可表示为 {{main.temp}}，对应 JSON 输入文档中的 main.temp 部分。

3. Handlebars单元测试

例 7-6 中的单元测试使用 Handlebars 模板将城市数据渲染为 HTML。

例 7-6 cities-weather-transform-test/test/handlebars-spec.js

```
'use strict';

/* 声明：城市天气数据由OpenWeatherMap API通过
   Creative Commons Share A Like许可证发布。
   为兼容json-server，数据经过了一定的修改。
   此操作并不意味着得到了许可方的背书。

   此代码通过Creative Commons Share A Like许可证发布。
*/

var expect = require('chai').expect;
var jsonfile = require('jsonfile');
var fs = require('fs');
var handlebars = require('handlebars');

describe('cities-handlebars', function() {
  var jsonCitiesFileName = null;
  var htmlTemplateFileName = null;

  beforeEach(function() {
    var baseDir = __dirname + '/../..';

    jsonCitiesFileName = baseDir + '/data/cities-weather-short.json';
    htmlTemplateFileName = baseDir +
      '/templates/transform-html.hbs';
  });


  it('should transform cities JSON data to HTML', function(done) {
    jsonfile.readFile(jsonCitiesFileName, function(readJsonFileError,
      jsonObj) {
      if (!readJsonFileError) {
        fs.readFile(htmlTemplateFileName, 'utf8', function(
          readTemplateFileError, templateFileData) {
          if (!readTemplateFileError) {
            var template = handlebars.compile(templateFileData);
            var html = template(jsonObj);

            console.log('\n\n\nHTML Output:\n' + html);
            done();
          } else {
            done(readTemplateFileError);
          }
```

```
            });
          } else {
            done(readJsonFileError);
          }
        });
      });
    });
```

除了以下差别，这一 Handlebars 单元测试与之前相应的 Mustache 单元测试基本相同。

- 无须将从 `fs.readFile()` 读取的 Handlebars 模板转换为字符串。
- 渲染模板的操作分为两步。
  - `handlebars.compile()` 对模板进行编译，并将结果保存到 `template` 变量中。
  - `template()` 将输入的 JSON 对象 `jsonObj` 渲染成 HTML。

使用 `npm test` 运行以上测试后，可以看到另一个与目标 HTML 文档相似的结果。

4. Handlebars在线测试器

TryHandlebars 和 Architect 是两个非常优秀的在线测试工具，可用于简化 Handlebars 模板的迭代开发与测试工作。

使用 TryHandlebars 时，可以将 Handlebars 模板和 JSON 粘贴到相应的文本输入框中，如图 7-2 所示。

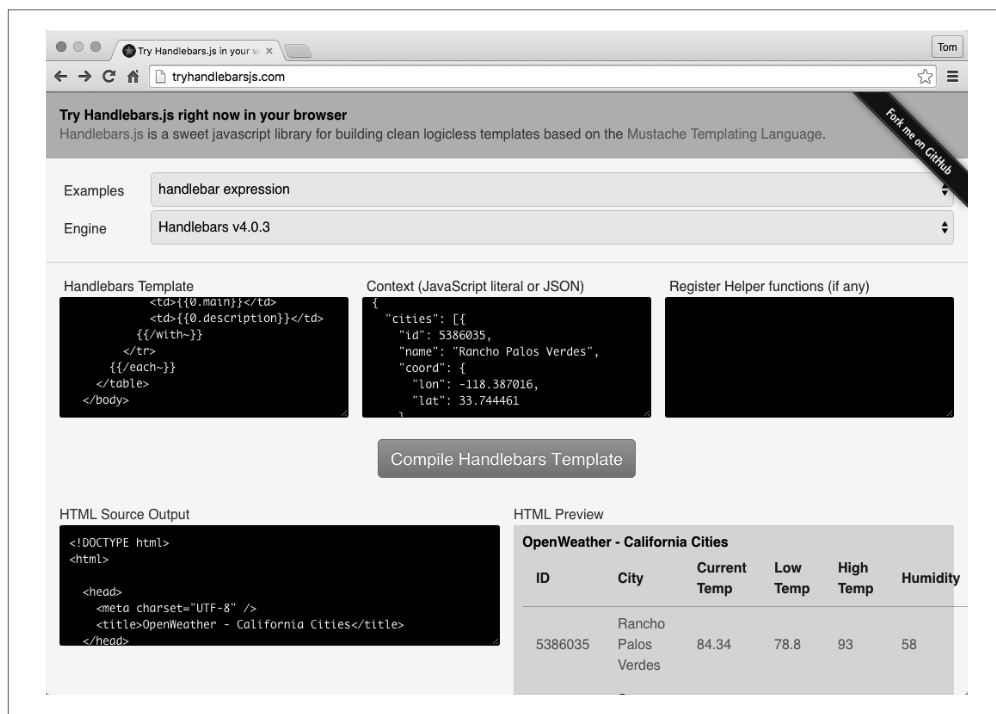


图 7-2：TryHandlebars.js：使用 Handlebars 将 JSON 转换为 HTML

除了 TryHandlebars，还可以使用 Architect 模板编辑器。在 Architect 的引擎下拉选框中选择 Handlebars.js，然后将 Handlebars 模板和输入的 JSON 分别粘贴到相应的区域中，即可看到如图 7-3 所示的结果。

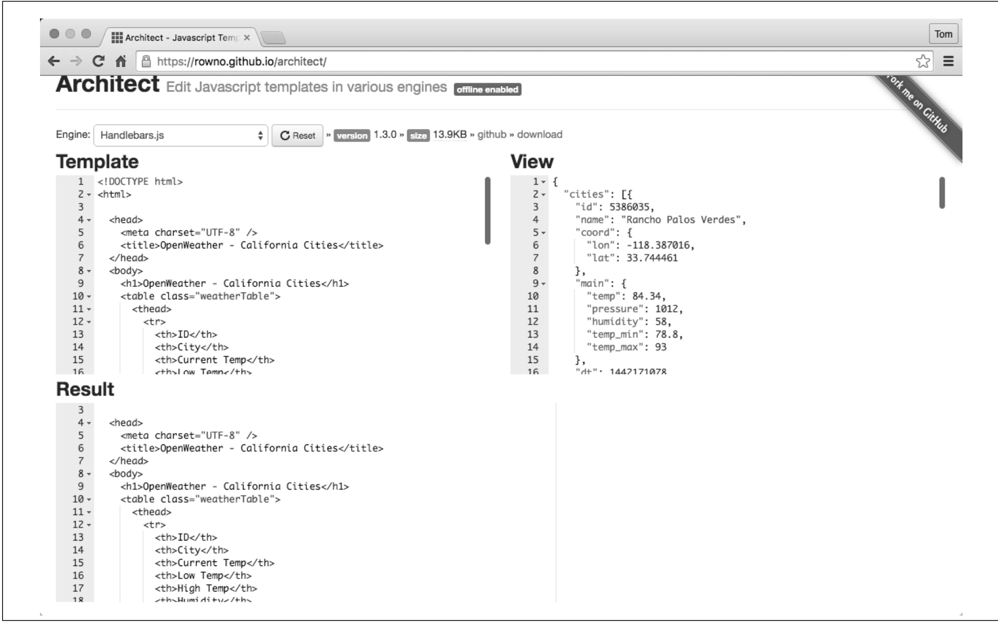


图 7-3：Architect：用 Handlebars 将 JSON 转换为 HTML

5. 在命令行中使用Handlebars

还可以在命令行中直接使用 Handlebars。如果已经安装 Node.js，则可以通过全局安装 `hb-interpolate` 模块（可在 GitHub 上找到该模块）来启用该操作：

```
npm install -g hb-interpolate

cd chapter-7

hb-interpolate -j ./data/cities-weather-short.json \
   -t ./templates/transform-html.hbs > output.html
```

6. 在其他平台上使用Handlebars

Handlebars 有着很好的跨平台支持。支持的平台包括：

- Node.js
- Ruby on Rails
- Java

7. Handlebars记分结果

基于本章开头的评估标准，表 7-2 展示了 Handlebars 的记分结果。

表7-2：Handlebars记分结果

关注度	Y
开发社区	Y
平台	JavaScript、Node.js、Java、Ruby on Rails
易于入门	Y
标准	N

Handlebars 是另一个广为使用的优秀引擎。与 Mustache 一样，Handlebars 不是一项标准，但它也提供了可靠的说明文档并能够跨平台工作。

7.4.4 转换工具评估——总结概要

对于将 JSON 转换为 HTML 的操作来说，Mustache 和 Handlebars 都是不错的选择，使用哪一个都可以满足需求。

介绍了将 JSON 转换为 HTML 的操作后，接下来本章将介绍 JSON 格式的转换。

7.5 JSON格式转换

只要在任何像样点的地方和 API 打过一段时间交道，你就应该意识到 API 并不是总能如预期那样工作。在设计 API 时，API 所返回的 JSON 响应经常成为最被忽视的一部分，而 API 所返回的数据很多时候也是极为难用的。即使返回的数据经过精心设计，你也可能只想读取其中一部分内容，或者需要将数据转换成另一种 JSON 结构以方便应用程序读取。

与第 6 章中的讨论类似，可以通过以下方法来解决这一问题。

- 解析 API 返回的 JSON，并程序化地操作结构体。
- 手工编写代码，将输入的 JSON 文档转换为另一种 JSON 结构。

可惜这些方案既繁琐又难用。事实上，现成的类库可以替你完成大部分工作，因此根本无须手工编写类似的工具代码。

7.5.1 存在的问题

在 JSON 格式转换操作上，我发现最大问题在于：标准缺失（无论是官方标准还是事实标准）。以之前章节中的 JSONPath 为例，虽然它不是官方标准，却是事实标准。作为一门查询语言，JSONPath 有着广泛的接受度，在多个平台上也都有自己的实现。但对于 JSON 转换类库来说，已有的类库大多是某个语言 / 平台上的单一实现，很难找到例外。我曾在技术社区中寻找过超越单个平台的、更加通用的产品。这个寻找最佳方案的过程是一次难忘的经历，但最后我却发现多个 JSON 转换类库并存比单一方案更好，希望这些类库能对你的项目产生帮助。

7.5.2 JSON格式转换类库

能够对 JSON 文档进行格式转换的类库有很多。本节将着重介绍以下几个类库：

- JSON Patch
- JSON-T
- Mustache
- Handlebars

先揭晓结论：Handlebars 是 JSON 格式转换的最佳选择（详见 7.5.8 节和 7.5.9 节）。接下来，我们将一一介绍这些 JSON 格式转换技术，阐明为何 Handlebars 是最佳选择。

7.5.3 其他优秀工具

用于 JSON 格式转换的类库有很多，本书无法详细介绍所有的类库。以下是另外 3 个值得了解的类库。

Jolt

　Jolt 只在 Java 环境中工作。

Json2Json

　Json2Json 只有 Node.js 版本。

`jsonapter`

　`jsonapter` 使用包含转换规则的外部模板文件，以声明式的风格转换 JSON 数据格式。外部模板文件与 XSL 比较像，但也仅止于比较像而已。`jsonapter` 及其模板规则都使用纯 JavaScript 来编写，XSL 则拥有自己的模板语言。遗憾的是，`jsonapter` 只在 JavaScript 和 Node.js 环境中工作。

7.5.4 目标JSON输出

关于输入数据，可参见 7.3 节中的内容。该数据的 `cities` 数组只包含 3 个元素，但即使是这样的数据，对我们的需求来说也过于复杂。我们不想使用所有的字段，因此需要执行以下步骤来简化数据结构。

- 保留 `cities` 数组中的 `id` 和 `name` 字段。
- 创建一个全新的、扁平的 `weather` 对象。
- 从其他结构中将有关天气的字段移到 `weather` 对象内。
  - `main.temp`、`main.humidity`、`main.temp_min`、`main.temp_max`
  - `wind.speed`
  - `weather.0.main` 和 `weather.0.description`
- 为简洁起见，修改字段名。

基于这些规则，处理后的输出应当如例 7-7 所示。

例 7-7　data/cities-weather-short-transformed.json

```
{
  "cities": [
    {
      "id": "5386035",
      "name": "Rancho Palos Verdes",
      "weather": {
```

```
        "currentTemp": 84.34,
        "lowTemp": 78.8,
        "hiTemp": 93,
        "humidity": 58,
        "windSpeed": 4.1,
        "summary": "Clear",
        "description": "Sky is Clear"
      }
    },
    {
      "id": "5392528",
      "name": "San Pedro",
      "weather": {
        "currentTemp": 84.02,
        "lowTemp": 78.8,
        "hiTemp": 91,
        "humidity": 58,
        "windSpeed": 4.1,
        "summary": "Clear",
        "description": "Sky is Clear"
      }
    },
    {
      "id": "3988392",
      "name": "Rosarito",
      "weather": {
        "currentTemp": 82.47,
        "lowTemp": 78.8,
        "hiTemp": 86,
        "humidity": 61,
        "windSpeed": 4.6,
        "summary": "Clouds",
        "description": "scattered clouds"
      }
    }
  ]
}
```

我们会使用不同的 JSON 格式转换类库将示例 JSON 输入数据转换为目标 JSON 输出，并根据操作的难易程度来评估每个 JSON 类库。

7.5.5 JSON Patch

作为一项 IETF 标准，JSON Patch 定义了一种数据格式，用于表示对资源的转换操作。JSON Patch 可以与 HTTP PATCH 标准协同工作。设计 HTTP PATCH 的目的在于修改由 API 所提供的资源。简而言之，HTTP PATCH 可以修改资源的**部分内容**，HTTP PUT 则用于**整体替换**资源。

设计 JSON Patch 的初衷是将其作为 HTTP 请求的一部分来考虑，与 HTTP 响应无关。因此，JSON Patch 其实是为 API 的提供者所设计的，考虑的并不是 API 使用者的需求。但本章是从 API 使用者的角度来描述的，因此我们会探讨使用 JSON Patch 对 HTTP 响应中的数据进行转换的操作。

1. JSON Patch语法

表 7-3 展示了能够用于处理 OpenWeatherMap 数据的一些 JSON Patch 操作。

表7-3：JSON Patch操作

JSON Patch操作	描述
添加 - `{ "op": "add", "path": "/wind", "value": { "direction": "W" } }`	在已存在的对象或数组中添加数据。不能使用该操作在文档中增加全新的对象
删除 - `{ "op": "remove", "path": "/main" }`	删除 `main` 对象
替换 - `{ "op": "replace", "path": "/weather/0/main", "value": "Rain" }`	替换文档中的某个值。该操作等价于先做 `remove`，然后再进行 `add`
复制 - `{ "op": "copy", "from": "/main/temp", "path": "/weather/0/temp" }`	将某个字段的值复制到另一处地方
移动 - `{ "op": "move", "from": "/main/temp", "path": "/weather/0/temp" }`	将 `main` 对象中的 `temp` 名称 – 值对移动到 `weather` 数组中

如需全面了解 JSON Patch，可访问其官方网站。JSON Patch 中的 `path` 值和 `from` 值均遵循了第 6 章中提及的 JSON Pointer 标准。

2. JSON Patch单元测试

例 7-8 中的单元测试展示了实际应用中的转换操作。这段代码使用了 JSON Patch 的 Node.js 模块。可访问该模块的 GitHub 主页获取更多信息。

该单元测试展示了如何使用 JSON Patch 将城市天气数据转换为目标 JSON 数据结构。

例 7-8 cities-weather-transform-test/test/json-patch-spec.json

```
'use strict';

/* 声明：城市天气数据由OpenWeatherMap API通过
   Creative Commons Share A Like许可证发布。
   为兼容json-server，数据经过了一定的修改。
   此操作并不意味着得到了许可方的背书。

   此代码通过Creative Commons Share A Like许可证发布。
*/

var expect = require('chai').expect;
var jsonfile = require('jsonfile');
var jsonpatch = require('json-patch');

var citiesTemplate = [
  {
    op: 'remove',
    path: '/coord'
  },
  {
    op: 'remove',
    path: '/dt'
  },
  {
    op: 'remove',
```

```
        path: '/clouds'
      },
      {
        op: 'remove',
        path: '/weather/0/id'
      },
      {
        op: 'remove',
        path: '/weather/0/icon'
      },
      {
        op: 'move',
        from: '/main/temp',
        path: '/weather/0/currentTemp'
      },
      {
        op: 'move',
        from: '/main/temp_min',
        path: '/weather/0/lowTemp'
      },
      {
        op: 'move',
        from: '/main/temp_max',
        path: '/weather/0/hiTemp'
      },
      {
        op: 'move',
        from: '/main/humidity',
        path: '/weather/0/humidity'
      },
      {
        op: 'move',
        from: '/weather/0/main',
        path: '/weather/0/summary'
      },
      {
        op: 'move',
        from: '/wind/speed',
        path: '/weather/0/windSpeed'
      },
      {
        op: 'remove',
        path: '/main'
      },
      {
        op: 'remove',
        path: '/wind'
      }
    ];

    describe('cities-json-patch', function() {
      var jsonFileName = null;
      var jsonCitiesFileName = null;

      beforeEach(function() {
```

```
    var baseDir = __dirname + '/../../data';

    jsonCitiesFileName = baseDir + '/cities-weather-short.json';
  });

  it('should patch all cities - fail', function(done) {
    jsonfile.readFile(jsonCitiesFileName, function(fileReadError,
      jsonObj) {
      if (!fileReadError) {
        try {
          var output = jsonpatch.apply(jsonObj, citiesTemplate);

          console.log('\n\n\n\Original JSON');
          console.log(jsonObj);
          console.log('\n\n\n\Patched JSON');
          console.log(JSON.stringify(output, null, 2));
          done();
        } catch (transformError) {
          console.error(transformError);
          done(transformError);
        }
      } else {
        console.error(fileReadError);
        done(fileReadError);
      }
    });
  });

  ...

});
```

在以上示例中，测试代码运行了一次 JSON Patch 的转换操作。可以在命令行中执行以下命令来运行该测试：

```
cd cities-weather-transform-test

npm test
```

正如接下来将看到的，`should patch all cities - fail` 这一测试用例会失败，结果如下所示：

```
cities-json-patch
{ [PatchConflictError: Value at coord does not exist]
message: 'Value at coord does not exist',
name: 'PatchConflictError' }
  1) should patch all cities - fail
```

在该测试用例中，因为内在的 JSON Pointer 规则只能识别对象，无法识别集合，所以 JSON Patch 无法找到路径为 /coord 的资源。

例 7-9 是另一个**几乎**能工作的单元测试。

例 7-9 cities-weather-transform-test/test/json-patch-spec.json

```
...

describe('cities-json-patch', function() {
  var jsonFileName = null;
  var jsonCitiesFileName = null;

  beforeEach(function() {
    var baseDir = __dirname + '/../../data';

    jsonCitiesFileName = baseDir + '/cities-weather-short.json';
  });

  ...

  it('should patch all cities - success (kind of)', function(done) {
    jsonfile.readFile(jsonCitiesFileName, function(fileReadError,
      jsonObj) {
      if (!fileReadError) {
        try {
          console.log('\n\n\n\Original JSON');
          console.log(jsonObj);
          var output = [];

          for (var i in jsonObj['cities']) {
            output.push(jsonpatch.apply(jsonObj['cities'][i],
              citiesTemplate));
          }

          console.log('\n\n\n\Patched JSON');
          console.log(JSON.stringify(output, null, 2));
          done();
        } catch (transformError) {
          console.error(transformError);
          done(transformError);
        }
      } else {
        console.error(fileReadError);
        done(fileReadError);
      }
    });
  });

});
```

虽然 `should patch all cities - success (kind of)` 这一测试用例能够成功运行，但由于以下原因，该测试并不能**有效**工作。

- 我们想要创建新的 `weather` 对象，而不仅仅只是使用已有的数组。遗憾的是，使用 JSON Patch 无法实现这一点。
- 该测试代码会遍历输入的 JSON，然后转换 `cities` 数组中的每个元素，最后再将所有的结果组装到 `output` 数组中。这么操作的原因是，JSON Patch 只能用于处理单个资源（对象），无法处理集合（数组）。

3. 在其他平台上使用JSON Patch

因为是一项标准，所以 JSON Patch 有着不错的跨平台支持。除 Node.js 外，JSON Patch 支持的平台还包括：

- Java
- Ruby

如需了解更多支持的平台和类库，可参考 JSON Patch 网站。

4. JSON Patch记分结果

基于本章开头的评估标准，表 7-4 展示了 JSON Patch 的记分结果。

表7-4：JSON Patch记分结果

关注度	Y
开发社区	Y
平台	JavaScript、Node.js、Java、Ruby on Rails
易于入门	N
标准	Y — RFC 6902

5. JSON Patch的局限性

JSON Patch 存在以下局限性。

- JSON Patch 不允许添加全新的数据结构，只允许对已有的结构和数据进行修改。
- JSON Patch 设计用于修改单个对象,没有考虑到对数组的处理问题。这么做的原因在于：JSON Patch 使用 JSON Pointer 来搜索数据，而 JSON Pointer 查询语句只能返回 JSON 文档中的单个字段。

虽然 JSON Patch 的初衷并不是对 HTTP 响应中的 JSON 数据进行转换，但这样的操作值得一试。JSON Patch 真正的初衷是与 HTTP PATCH 配合使用，对 HTTP 请求中的资源数据进行部分修改，用 JSON 来声明具体的修改规则。

相比于 JSON Patch，还可以使用更好的类库将一个 JSON 转换为另一种 JSON 数据结构。接下来我们尝试一下 JSON-T。

7.5.6 JSON-T

JSON-T 由 JSONPath 的创建者 Stefan Goessner 于 2006 年开发，是早期的 JSON 转换类库之一。JSON-T 与 XML 中的 XSLT 比较像，都使用了包含转换规则的模板文件。

1. JSON-T语法

JSON-T 使用在 JavaScript 对象字面量中定义的转换规则，其中每条规则都和对象中的一个名称 – 值对对应。规则的形式如下：

```
var transformRules = {
  'ruleName': 'transformationRule',
  'ruleName': function
  ...
};
```

对于以上形式，需要注意以下几点。

- 每个 `ruleName` 或 `transformationRule` 都必须由单引号（`''`）或双引号（`""`）括起来。
- 每个 `transformationRule` 都有由大括号括起来的一条或多条转换表达式，如 `{cities}`。
- 转换表达式可解析为另一个 `ruleName` 或文档中的某个字段，即数组、对象或名称 – 值对。

以下示例展示了用于转换 OpenWeatherMap 数据的 JSON-T 规则：

```
var transformRules = {
  'self': '{ "cities": [{cities}] }',
  'cities[*]': '{ "id": "{$.id}", "name": "{$.name}", ' +
    '"weather": { "currentTemp": {$.main.temp}, "lowTemp": {$.main.temp_min}, ' +
    '"hiTemp": {$.main.temp_max}, "humidity": {$.main.humidity}, ' +
    '"windSpeed": {$.wind.speed}, "summary": "{$.weather[0].main}", ' +
    '"description": "{$.weather[0].description}" } },'
};
```

该示例的工作机制如下。

- `self` 是声明新的 JSON 文档格式的最高级规则，其中 `{cities}` 会引用 `cities[*]` 规则的返回结果。
- `cities[*]` 规则声明如何对 `cities` 数组进行格式化。
  - `cities[*]` 规则中的星号表示该规则作用于 `cities` 数组元素。
  - `*` 解析为每个数组的索引。
  - `{$.}` 是一种快捷表示法。`{$.name}` 规则告诉 JSON-T 从每个 `cities` 数组元素的 `name` 字段中抽取数据，完整的表示则为：`cities[*].name`。

可以访问 JSON-T 官方网站中的"Basic Rules"部分来了解所有的转换规则文档。

2. JSON-T单元测试

例 7-10 中的单元测试使用 `jsont` 这一 Node.js 模块来展示 JSON-T 的使用。

例 7-10　cities-weather-transform-test/test/jsont-spec.js

```
'use strict';

/* 声明：城市天气数据由OpenWeatherMap API通过
   Creative Commons Share A Like许可证发布。
   为兼容json-server，数据经过了一定的修改。
   此操作并不意味着得到了许可方的背书。

   此代码通过Creative Commons Share A Like许可证发布。
*/

var expect = require('chai').expect;
var jsonfile = require('jsonfile');
var jsonT = require('../lib/jsont').jsonT;

describe('cities-jsont', function() {
  var jsonCitiesFileName = null;

  var transformRules = {
    'self': '{ "cities": [{cities}] }',
    'cities[*]': '{ "id": "{$.id}", "name": "{$.name}", ' +
```

```
      '"weather": { "currentTemp": {$.main.temp}, "lowTemp": {$.main.temp_min}, ' +
      '"hiTemp": {$.main.temp_max}, "humidity": {$.main.humidity}, ' +
      '"windSpeed": {$.wind.speed}, "summary": "{$.weather[0].main}", ' +
      '"description": "{$.weather[0].description}" } },'
  };

  ...

  beforeEach(function() {
    var baseDir = __dirname + '/../../data';
    jsonCitiesFileName = baseDir + '/cities-weather-short.json';
  });

  it('should transform cities JSON data', function(done) {
    jsonfile.readFile(jsonCitiesFileName, function(readFileError,
      jsonObj) {
      if (!readFileError) {
        var jsonStr = jsonT(jsonObj, transformRules);

        jsonStr = repairJson(jsonStr);
        console.log(JSON.stringify(JSON.parse(jsonStr), null, 2));
        done();
      } else {
        done(readFileError);
      }
    });
  });
});
```

注意，以上测试代码调用 `repairJson()` 函数来生成合法的 JSON：

```
function repairJson(jsonStr) {
  var repairedJsonStr = jsonStr;

  var repairs = [
    [/,\s*}/gi, ' }'],
    [/,\s*\]/gi, ' ]']
  ];

  for (var i = 0, len = repairs.length; i < len; ++i) {
    repairedJsonStr = repairedJsonStr.replace(repairs[i][0], repairs[i][1]);
  }

  return repairedJsonStr;
}

// 在测试用例中做如下修改：

...
jsonStr = repairJson(jsonStr);
console.log(JSON.stringify(JSON.parse(jsonStr), null, 2));
...
```

如果不做任何修改，则 JSON-T 会在 `cities` 数组的最后一个元素后面添加逗号，导致转换后的 JSON 非法。为了修复这一问题，示例中的 `repairJson()` 函数使用了正则表达式来删除闭合大括号（`}`）或闭合数组符号（`]`）前的逗号。虽然大多数编程语言都支持正则表达

式，但这种自行编写代码来纠正输出的做法是比较糟糕的。你不应该再重复编写基础设施代码。

3. 在其他平台上使用JSON-T

除了 Node.js，JSON-T 还可以在以下平台上运行。

浏览器
  JSON-T 能够以单个 JavaScript 文件 jsont.js 的形式运行。

Ruby
  JSON-T 存在纯 Ruby 的实现。

我未能找到 JSON-T 的纯 Java 实现。

4. JSON-T记分结果

基于本章开头的评估标准，表 7-5 展示了 JSON-T 的记分结果。

表7-5：JSON-T记分结果

关注度	Y
开发社区	Y
平台	JavaScript、Node.js、Ruby on Rails
易于入门	N
标准	N

5. JSON-T的局限性

JSON-T 存在以下局限性。

- 过于复杂的语法。
- 没有 Java 实现。
- 无法处理字符串中的转义文本。例如，JSON-T 会将字符串 `"escapedString": "I have a \"string within\" a string"` 转换为 `"escapedString": "I have a "string within " a string"` 这样的非法字符串，导致必须使用正则表达式来修复结果。
- 处理数组或对象的最后一个元素有误。

因为能够处理整个 JSON 文档，所以 JSON-T 相较于 JSON Patch 有了小小的改进；但如果想要正常工作的话，JSON-T 依旧需要开发人员编写额外的代码。JSON-T 往正确的方向前进了一步，但在实际的开发环境中并未达到可以应用的程度。JSON-T 擅长将 JSON 转换为 HTML，但其设计初衷并不包括对 JSON 文档的结构进行转换。

接下来我们看一下 Mustache。

7.5.7 Mustache

在前面的内容中，我们看到 Mustache 可以轻松地将 JSON 转换为 HTML。接下来的内容将展示如何使用 Mustache 将城市数据转换为目标 JSON 文档。

例 7-11 是一个进行转换的 Mustache 模板（7.4 节中介绍过 Mustache 模板的详细信息）。

例 7-11　templates/transform-json.mustache

```
{
  "cities": [
    {{#cities}}
      {
        "id": "{{id}}",
        "name": "{{name}}",
        "weather": {
          {{#main}}
          "currentTemp": {{temp}},
          "lowTemp": {{temp_min}},
          "hiTemp": {{temp_max}},
          "humidity": {{humidity}},
          {{/main}}
          "windSpeed": {{wind.speed}},
          {{#weather.0}}
          "summary": "{{main}}"
          "description": "{{description}}"
          {{/weather.0}}
        }
      },
    {{/cities}}
  ]
}
```

在 Architect 模板编辑器中运行该模板。在 Engine 下拉菜单中选择 Mustache.js，然后将 Mustache 模板和输入的 JSON 粘贴到相应的文本框中。可以看到如图 7-4 所示的结果。

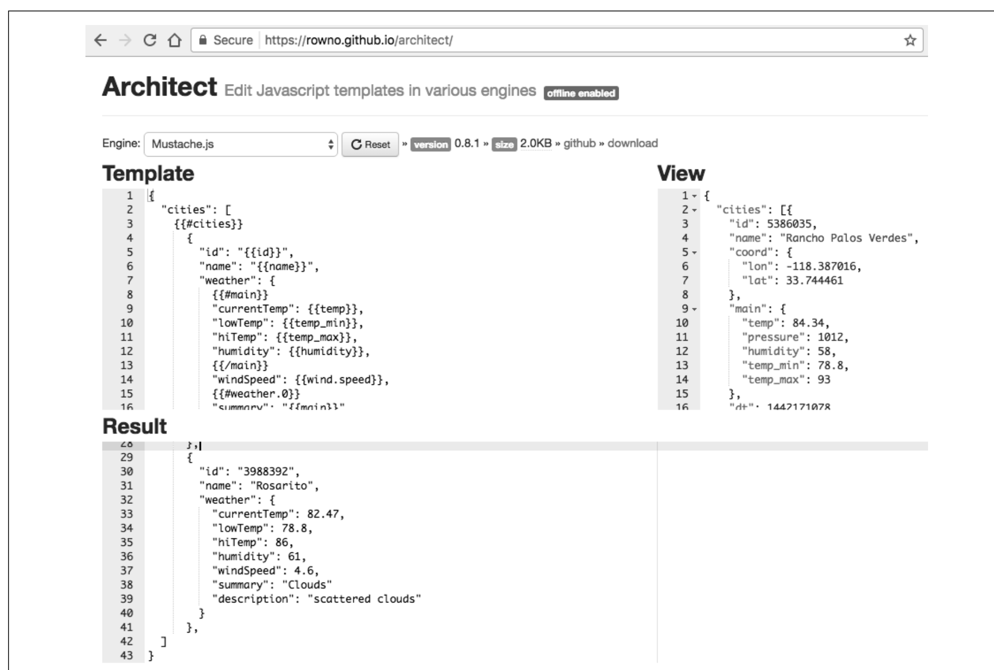


图 7-4：在 Architect 中用 Mustache 转换 JSON

观察 Result 文本框中的 JSON 处理结果的第 41 行，我们可以看到多余的逗号，而这会导致 JSON 结果非法。如果将该 JSON 结果粘贴到 JSONLint 中，就可以看到该 JSON 确实是非法的：

```
Results

Error: Parse error on line 11:
...ummary": "Clear"                    "description": "Sky
---------------------^
Expecting 'EOF', '}', ':', ',', ']', got 'STRING'
```

Mustache的局限性

与 JSON-T 一样，在处理 JSON 输入时，Mustache 无法确定当前所处理的元素是否为数组或对象中的最后一个元素。因此，Mustache 不能完美地用于处理 JSON 结构转换。

接下来我们开始介绍 Handlebars。

7.5.8 Handlebars

正如我们之前所看到的，Handlebars 可以很好地将 JSON 转换为 HTML，例 7-12 中的模板可用于将城市 JSON 数据转换为目标 JSON 输出。

例 7-12 templates/transform-json.hbs

```
{
  "cities": [
    {{#each cities}}
      {
        "id": "{{id}}",
        "name": "{{name}}",
        "weather": {
          {{#main}}
          "currentTemp": {{temp}},
          "lowTemp": {{temp_min}},
          "hiTemp": {{temp_max}},
          "humidity": {{humidity}},
          {{/main}}
          "windSpeed": {{wind.speed}},
          {{#each weather}}
          "summary": "{{main}}",
          "description": "{{description}}"
          {{/each}}
        }
      }{{#unless @last}},{{/unless}}
    {{/each}}
  ]
}
```

除了一处明显的差异，该模板与 7.4 节中的 Handlebars 所用的模板很相似。以下这行代码准确地实现了我们的需求：**除了**最后一个元素，在所有元素的后面添加一个逗号：

```
{{#unless @last}},{{/unless}}
```

以下是对这行代码的解释。

- `{{#unless}}` 是 Handlebars 的内置辅助指令，仅当其条件语句返回 `false` 时才渲染内含的代码块。
- `@last` 是 Handlebars 的内置变量，只在当前元素是数组中的最后一个元素时才返回 `true`。

如需了解更多有关 `{{#unless}}` 和 `@last` 的信息，可参考 Handlebars 官方网站。

在 Architect 模板编辑器中运行该模板。在 Engine 下拉菜单中选择 Handlebars.js，然后将 Handlebars 模板和输入的 JSON 粘贴到相应的文本框中。可以看到如图 7-5 所示的结果。

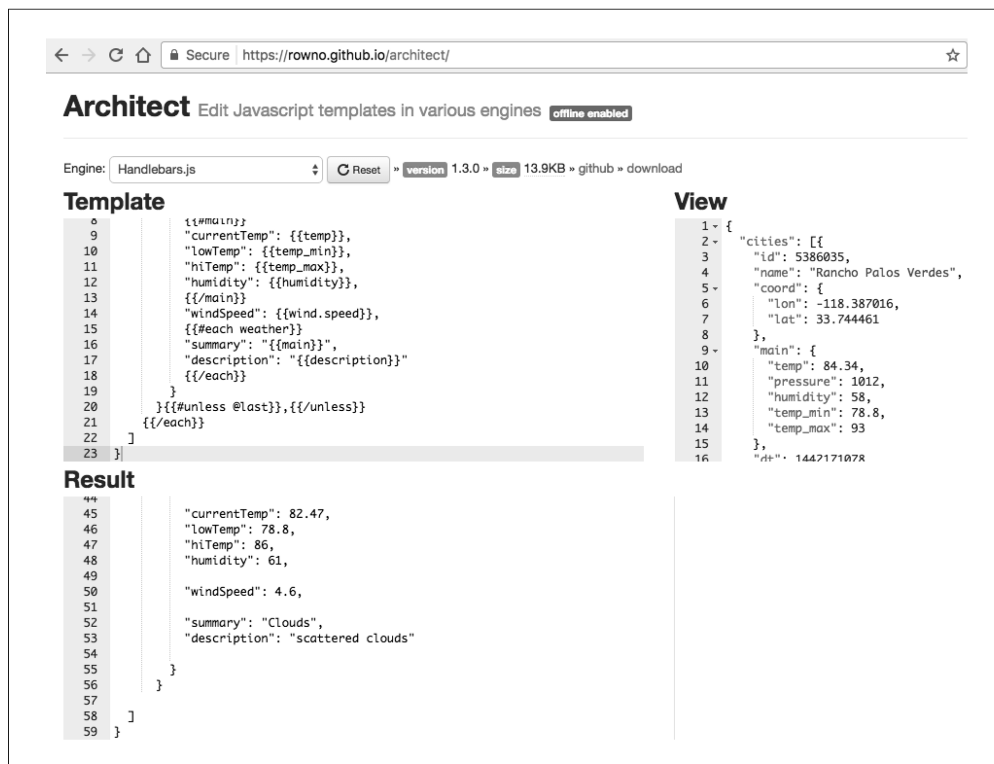


图 7-5：在 Architect 中用 Handlebars 转换 JSON

观察 Result 文本框中的 JSON 处理结果的第 56 行，我们可以看到多余的逗号消失了，因此处理后的 JSON 是合法的。如果将这一 JSON 结果粘贴到 JSONLint 中，可以看到该 JSON 确实是合法的，如图 7-6 所示。

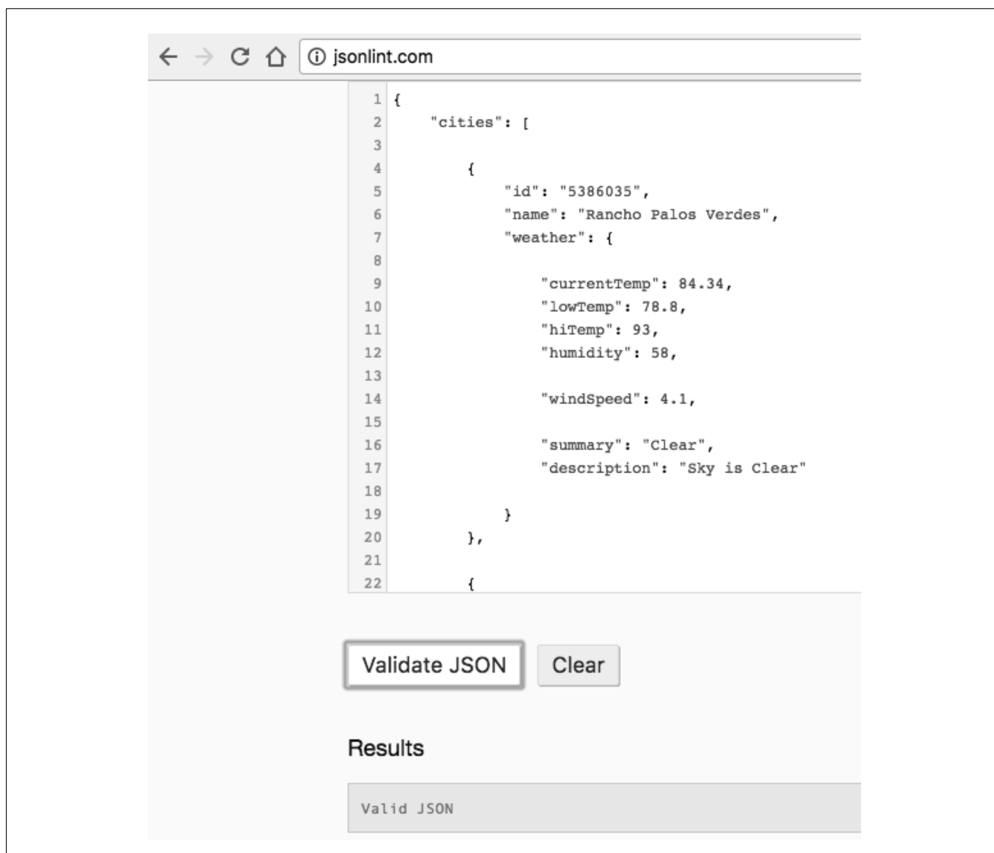


图 7-6：用 JSONLint 校验经过 Handlebars 处理后的 JSON 结果

这样的结果正是我们所探寻的。就像我们之前所提到的那样，Handlebars 与 Mustache 的区别在于：Handlebars 中存在足量的控制逻辑，能够使得 JSON 结构转换正常进行。

7.5.9 转换工具评估——总结概要

根据评估标准和总体的可用性，Handlebars 是我在 JSON 结构转换领域最为偏爱的工具，原因有以下几点。

- Handlebars 是唯一一个不用编写额外处理代码的类库。其内置的条件逻辑使得这一点成为可能。
- 跨平台支持性较好。
- 模板语言丰富，能够满足大多数的转换需求。
- 它是声明式的，但也支持在自定义辅助指令中编写额外的逻辑代码。
- 优秀的在线工具使得开发工作变得更加便捷。

介绍了 JSON 结构转换后，接下来本章将描述从 JSON 转换为 XML 的操作。

7.6 JSON与XML的相互转换

开发者和架构师经常会碰到需要与遗留系统进行整合的情况，而这些遗留系统可能仍在使用 XML。为了实现完美的关注点分离，在自己的系统边缘增加薄薄的适配层以负责对 XML 和 JSON 进行相互转换，无疑是十分重要的。

7.6.1 传统转换工具

在 XML 元素（如 `<weather>`）和 JSON 之间进行相互转换是一件比较简单的事情，但在 XML 属性和 JSON 之间进行相互转换就比较困难了。因为“属性的表示”在 JSON 中缺乏标准，所以 XML 属性和 JSON 之间的相互转换是一种有损转换，即无法将转换后的 JSON 重新转换成初始的 XML，反之亦然。切记，JSON 的核心组成结构是对象、数组和名称 - 值对。

比如，XML 属性用于提供描述元素的元数据，大致如下所示：

```
<weather temp="84.34" pressure="1012" humidity="58"
         temp_min="78.8" temp_max="93"/>
```

在这一 XML 片段中，`temp`、`pressure`、`humidity`、`temp_min` 和 `temp_max` 属性描述了 `weather` 元素。在过去 XML 还很流行的时候（约 1998~2008 年），XML Schema 的很多设计人员将 XML 属性用于：

- 减少在网络上传输的消息的总体大小；
- 简化 XML 和平台（如 Java、JavaScript、Ruby 或 C#）之间的转换工作。

我们将介绍如何在 XML 和 JSON 之间进行直接转换，以下是一些知名的传统转换工具：

- Badgerfish
- Parker
- JsonML
- Spark
- GData
- Abdera

因为 Badgerfish 和 Parker 的知名度较高，所以本章将主要介绍这两个工具。详尽谈论、深入比较上述 XML-JSON 相互转换的工具超出了本书的探讨范围，相关详情可参考 the Open311 wiki。

为了比较 Badgerfish 和 Parker，我们先展示一个基于 OpenWeatherMap 数据的示例 XML 文档。然后，我们将比较这两个工具是如何将 XML 转换成 JSON 的。例 7-13 展示了输入的 XML。

例 7-13 data/cities-weather-short.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<cities>
  <city>
    <id>5386035</id>
    <name>Rancho Palos Verdes</name>
```

```
    &lt;coord&gt;
      &lt;lon&gt;-118.387016&lt;/lon&gt;
      &lt;lat&gt;33.744461&lt;/lat&gt;
    &lt;/coord&gt;
    &lt;main temp="84.34" pressure="1012" humidity="58" temp_min="78.8" temp_max="93"/&gt;
    <dt>1442171078</dt>
    &lt;wind&gt;
      &lt;speed&gt;4.1&lt;/speed&gt;
      &lt;deg&gt;300&lt;/deg&gt;
    &lt;/wind&gt;
    &lt;clouds&gt;
      &lt;all&gt;5&lt;/all&gt;
    &lt;/clouds&gt;
    &lt;weather&gt;
      &lt;id&gt;800&lt;/id&gt;
      &lt;main&gt;Clear&lt;/main&gt;
      &lt;description&gt;Sky is Clear&lt;/description&gt;
      &lt;icon&gt;02d&lt;/icon&gt;
    &lt;/weather&gt;
  &lt;/city&gt;
  &lt;city&gt;
    &lt;id&gt;5392528&lt;/id&gt;
    &lt;name&gt;San Pedro&lt;/name&gt;
    &lt;coord&gt;
      &lt;lon&gt;-118.29229&lt;/lon&gt;
      &lt;lat&gt;33.735851&lt;/lat&gt;
    &lt;/coord&gt;
    &lt;main temp="84.02" pressure="1012" humidity="58" temp_min="78.8" temp_max="91"/&gt;
    <dt>1442171080</dt>
    &lt;wind&gt;
      &lt;speed&gt;4.1&lt;/speed&gt;
      &lt;deg&gt;300&lt;/deg&gt;
    &lt;/wind&gt;
    &lt;clouds&gt;
        &lt;all&gt;5&lt;/all&gt;
    &lt;/clouds&gt;
    &lt;weather&gt;
      &lt;id&gt;800&lt;/id&gt;
      &lt;main&gt;Clear&lt;/main&gt;
      &lt;description&gt;Sky is Clear&lt;/description&gt;
      &lt;icon&gt;02d&lt;/icon&gt;
    &lt;/weather&gt;
  &lt;/city&gt;
  &lt;city&gt;
    &lt;id&gt;3988392&lt;/id&gt;
    &lt;name&gt;Rosarito&lt;/name&gt;
    &lt;coord&gt;
      &lt;lon&gt;-117.033333&lt;/lon&gt;
      &lt;lat&gt;32.333328&lt;/lat&gt;
    &lt;/coord&gt;
    &lt;main temp="82.47" pressure="1012" humidity="61" temp_min="78.8" temp_max="86"/&gt;
    <dt>1442170905</dt>
    &lt;wind&gt;
      &lt;speed&gt;4.6&lt;/speed&gt;
      &lt;deg&gt;240&lt;/deg&gt;
```

```
      </wind>
      <clouds>
        <all>32</all>
      </clouds>
      <weather>
        <id>802</id>
        <main>Clouds</main>
        <description>scattered clouds</description>
        <icon>03d</icon>
      </weather>
    </city>
  </cities>
```

1. Badgerfish

Badgerfish 提供了非常优秀的在线测试器，能有效简化将输入的 XML 转换为 JSON 的操作。图 7-7 展示了 Badgerfish Online Tester。

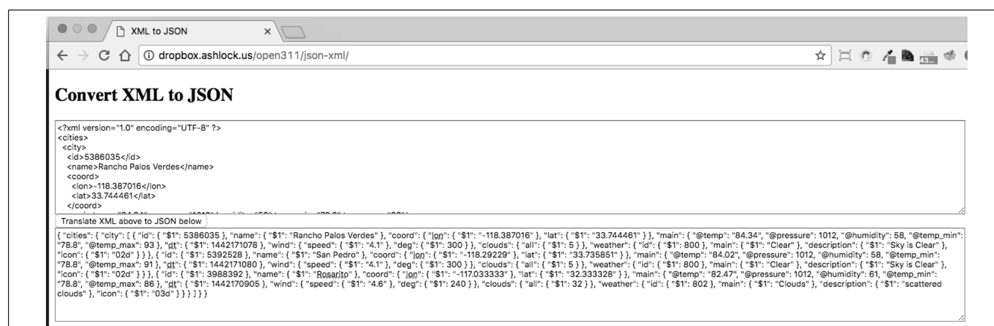


图 7-7：Badgerfish 在线测试器——将 XML 转换为 JSON

将输入的 XML 粘贴到 Convert XML to JSON 标题下的文本输入框中，点击 Translate XML above to JSON below button 按钮，即可在结果文本框中看到非常紧凑的 JSON。可以使用 JSONLint 或包含 JSON 优化显示插件的任何文本编辑器，处理后即可看到更具可读性的 JSON 结果，如例 7-14 所示。

例 7-14 data/cities-weather-short-badgerfish.json

```
{
  "cities": {
    "city": [{
      "id": {
        "$1": 5386035
      },
      "name": {
        "$1": "Rancho Palos Verdes"
      },
      "coord": {
        "lon": {
          "$1": "-118.387016"
        },
        "lat": {
          "$1": "33.744461"
```

```
      }
    },
    "main": {
      "@temp": "84.34",
      "@pressure": 1012,
      "@humidity": 58,
      "@temp_min": "78.8",
      "@temp_max": 93
    },
    "dt": {
      "$1": 1442171078
    },
    "wind": {
      "speed": {
        "$1": "4.1"
      },
      "deg": {
        "$1": 300
      }
    },
    "clouds": {
      "all": {
        "$1": 5
      }
    },
    "weather": {
      "id": {
        "$1": 800
      },
      "main": {
        "$1": "Clear"
      },
      "description": {
        "$1": "Sky is Clear"
      },
      "icon": {
        "$1": "02d"
      }
    }
  }, {
    "id": {
      "$1": 5392528
    },
    "name": {
      "$1": "San Pedro"
    },
    "coord": {
      "lon": {
        "$1": "-118.29229"
      },
      "lat": {
        "$1": "33.735851"
      }
    },
    "main": {
```

```
      "@temp": "84.02",
      "@pressure": 1012,
      "@humidity": 58,
      "@temp_min": "78.8",
      "@temp_max": 91
    },
    "dt": {
      "$1": 1442171080
    },
    "wind": {
      "speed": {
        "$1": "4.1"
      },
      "deg": {
        "$1": 300
      }
    },
    "clouds": {
      "all": {
        "$1": 5
      }
    },
    "weather": {
      "id": {
        "$1": 800
      },
      "main": {
        "$1": "Clear"
      },
      "description": {
        "$1": "Sky is Clear"
      },
      "icon": {
        "$1": "02d"
      }
    }
  }, {
    "id": {
      "$1": 3988392
    },
    "name": {
      "$1": "Rosarito"
    },
    "coord": {
      "lon": {
        "$1": "-117.033333"
      },
      "lat": {
        "$1": "32.333328"
      }
    },
    "main": {
      "@temp": "82.47",
      "@pressure": 1012,
      "@humidity": 61,
```

```
            "@temp_min": "78.8",
            "@temp_max": 86
          },
          "dt": {
            "$1": 1442170905
          },
          "wind": {
            "speed": {
              "$1": "4.6"
            },
            "deg": {
              "$1": 240
            }
          },
          "clouds": {
            "all": {
              "$1": 32
            }
          },
          "weather": {
            "id": {
              "$1": 802
            },
            "main": {
              "$1": "Clouds"
            },
            "description": {
              "$1": "scattered clouds"
            },
            "icon": {
              "$1": "03d"
            }
          }
        }]
      }
    }
```

Badgerfish 的核心规则包括以下几点。

- XML 元素名会成为 JSON 对象的属性名。
- XML 元素的文本内容会成为相应 JSON 对象的 `$` 属性值。例如，`<name>Rancho Palos Verdes</name>` 会转换为 `"name": { "$1": "Rancho Palos Verdes" }`。
- 内嵌元素会成为对象的内嵌属性。例如，以下 XML：

```
<wind>
  <speed>4.1</speed>
  <deg>300</deg>
</wind>
```

会转换为

```
"wind": {
  "speed": {
    "$1": "4.1"
```

```
  },
  "deg": {
    "$1": 300
  }
}
```

- 如果同一层级中存在多个名称相同的元素，则这些元素会转换为数组元素。以下 XML：

  ```
  <city>
  </city>
  <city>
  </city>
  ```

  会转换为

  ```
  "city":[ { ... } ]
  ```

- XML 元素的属性会转换为以 @ 符号开头的对象字段名。例如，以下 XML：

  ```
  <main temp="84.02" pressure="1012" humidity="58"
        temp_min="78.8" temp_max="91"/>
  ```

  会转换为

  ```
  "main": {
    "@temp": "84.34",
    "@pressure": 1012,
    "@humidity": 58,
    "@temp_min": "78.8",
    "@temp_max": 93
  }
  ```

以上描述忽略了很多细节。事实上，Badgerfish 提供了非常优秀的文档资源。如需了解更多信息，可参考：

- Badgerfish 官方网站；
- Badgerfish 文档；
- Badgerfish 在线测试器。

2. Parker

Parker 提供的转换方法非常简单，但它直接忽略了 XML 属性，因此使用 Parker 将 XML 转换为 JSON 会损失属性信息。根据输入的 XML，采用 Parker 得到的 JSON 转换结果如例 7-15 所示。

例 7-15　data/cities-weather-short-parker.json

```
{
  "cities": [{
    "id": 5386035,
    "name": "Rancho Palos Verdes",
    "coord": {
      "lon": -118.387016,
      "lat": 33.744461
    },
    "main": null,
```

```
    "dt": 1442171078,
    "wind": {
      "speed": 4.1,
      "deg": 300
    },
    "clouds": {
      "all": 5
    },
    "weather": [{
      "id": 800,
      "main": "Clear",
      "description": "Sky is Clear",
      "icon": "02d"
    }]
  }, {
    "id": 5392528,
    "name": "San Pedro",
    "coord": {
      "lon": -118.29229,
      "lat": 33.735851
    },
    "main": null,
    "dt": 1442171080,
    "wind": {
      "speed": 4.1,
      "deg": 300
    },
    "clouds": {
      "all": 5
    },
    "weather": [{
      "id": 800,
      "main": "Clear",
      "description": "Sky is Clear",
      "icon": "02d"
    }]
  }, {
    "id": 3988392,
    "name": "Rosarito",
    "coord": {
      "lon": -117.033333,
      "lat": 32.333328
    },
    "main": null,
    "dt": 1442170905,
    "wind": {
      "speed": 4.6,
      "deg": 240
    },
    "clouds": {
      "all": 32
    },
    "weather": [{
      "id": 802,
      "main": "Clouds",
```

```
        "description": "scattered clouds",
        "icon": "03d"
      }]
    }]
  }
```

Parker 的核心规则包括以下几点。

- XML 元素名会成为 JSON 对象的属性名。
- 忽略属性。
- 内嵌元素转换为对象的内嵌属性。

Parker 很简单，但存在以下问题。

- 转换过程是有信息损耗的，当使用 Parker 将 XML 转换为 JSON 时，XML 属性会被忽略。
- 缺乏文档和工具支持。

7.6.2 传统转换工具所面对的问题

之前列举的 XML-JSON 相互转换工具存在以下局限。

- 没有一个工具成为公认的标准而广为接受。
- 缺乏跨平台支持和完整实现。
- 很多工具文档不全。
- 有些工具进行数据转换时会损失信息（Parker）。
- 有些工具进行数据转换时会改变数据结构（Badgerfish）。

7.6.3 XML-JSON相互转换——总结概要

了解了这些传统转换工具的缺点后，我建议在进行以下转换时避免使用上述工具。

将 XML 转换为 JSON

在当前编程平台上，使用知名类库将 XML 解析为对象 /Hash（我们将在 Node.js 示例中使用 `xml2js`）。然后使用 `JSON.stringify()`（如果使用的是 JavaScript）将该对象 /Hash 转换为 JSON。第 3 章和第 4 章分别展示了如何将 Ruby 数据和 Java 数据转换为 JSON。

将 JSON 转换为 XML

在当前编程平台上，使用常用的类库将 JSON 解析为相应的数据结构。对于 JavaScript 来说，`JSON.parse()` 就足够了。第 3 章和第 4 章分别展示了如何将 JSON 解析为 Ruby 和 Java 数据。然后根据该数据结构来生成 XML 文档（该过程称为 marshaling）。在后面基于 Node.js 的 Mocha/Chai 单元测试中，我们依旧使用 `xml2js` 来完成这一操作。

不要纠结于具体的转换工具 / 风格，应当关注以下重点。

- 选择最有利于自己目标的方案。
- 使用已经熟悉且上手的类库。
- 测试转换结果，确保没有损失任何数据。
- 保持简洁。

- 封装所有的转换操作，并确保其与目前的企业级应用程序架构兼容。

简而言之，选择当前编程平台中最好的类库，然后修复或绕过其缺陷。

解析/生成XML类库

XML 已经存在很长一段时间了，每个主流平台上也都有可靠的解决方案，如下所示。

Node.js
: 我们将使用 `xml2js`。

Ruby
: Ruby 中有不少不错的类库，最好的两个是 LibXml 和 Nokogiri。

Java
: 在 Java 社区中，Java Architecture for XML Binding 多年来一直是相关领域的主要选择。

7.6.4　JSON-XML相互转换——单元测试

例 7-16 中的单元测试套件对"将 JSON 转换为 XML"与"将 XML 转换为 JSON"进行了测试，其中用到了以下技术。

`xmL2js`
: 可以使用 `xml2js` 在 XML 和 JavaScript 数据结构之间进行相互转换，相关详情可参考其 GitHub 主页。

`JSON.parse()/JSON.stringify()`
: 二者可用于在 JSON 和 JavaScript 数据结构之间进行相互转换。如需了解更多有关 `JSON.parse()/JSON.stringify()` 的信息，可参考 MDN 以及第 3 章中的相关内容。

例 7-16　cities-weather-transform-test/test/json-xml-spec.js

```
'use strict';

/* 声明：城市天气数据由OpenWeatherMap API通过
   Creative Commons Share A Like许可证发布。
   为兼容json-server，数据经过了一定的修改。
   此操作并不意味着得到了许可方的背书。

   此代码通过Creative Commons Share A Like许可证发布。
*/

var expect = require('chai').expect;
var jsonfile = require('jsonfile');
var fs = require('fs');
var xml2js = require('xml2js');

describe('json-xml', function() {
  var jsonCitiesFileName = null;
  var xmlCitiesFileName = null;

  beforeEach(function() {
    var baseDir = __dirname + '/../..';
```

```
      jsonCitiesFileName = baseDir + '/data/cities-weather-short.json';
      xmlCitiesFileName = baseDir +
        '/data/cities-weather-short.xml';
    });

    it('should transform cities JSON data to XML', function(done) {
      jsonfile.readFile(jsonCitiesFileName, function(readJsonFileError,
        jsonObj) {
        if (!readJsonFileError) {
          var builder = new xml2js.Builder();
          var xml = builder.buildObject(jsonObj);
          console.log('\n\n\nXML Output:\n' + xml);
          done();
        } else {
          done(readJsonFileError);
        }
      });
    });

    it('should transform cities XML data to JSON', function(done) {
      fs.readFile(xmlCitiesFileName, 'utf8', function(
        readXmlFileError, xmlData) {
        if (!readXmlFileError) {
          var parser = new xml2js.Parser();

          parser.parseString(xmlData, function(error, xmlObj) {
            if (!error) {
              console.log('\n\n\nJSON Output:\n' +
                JSON.stringify(xmlObj, null, 2));

              done();
            } else {
              done(error);
            }
          });
        } else {
          done(readXmlFileError);
        }
      });
    });
  });
```

这段代码的工作机制如下。

- `beforeEach()` 会在每个单元测试用例运行前先执行，并完成测试所需的配置工作。在本例中，该方法会拼接生成输入的 JSON 文件和输出的 XML 文件的文件名。
- 在 `'should transform cities JSON data to XML'` 单元测试用例中：
  - `jsonfile.readFile()` 读取输入的 JSON 文件，并将其解析为 JavaScript 对象（`jsonObj`）；
  - `xml2js.Builder()` 创建并返回一个对象，该对象可将 JSON 转换为 XML；
  - `builder.buildObject(jsonObj)` 将从 JSON 文件中读取的 JavaScript 对象转换为 XML 字符串。
- 在 `'should transform cities XML data to JSON'` 单元测试用例中：

- `fs.readFile()` 读取 XML 文件，并将结果存储在字符串中；
- `xml2js.Parser()` 创建并返回一个 XML 解析器；
- `parser.parseString()` 将从 XML 文件中读取的 XML 字符串转换为 JavaScript 对象（`xmlObj`）；
- `JSON.stringify()` 将 `xmlObj` 这一 JavaScript 对象转换为 JSON 字符串。

7.7 本章回顾

本章介绍了用于执行以下操作的一些 JSON 转换类库。

- 将 JSON 转换为 HTML。
  - Mustache 或者 JSON 都可以很好地实现这一操作。
- 将 JSON 转换为更加整洁的另一种 JSON 结构。
  - 首选 Handlebars。
- 在 XML 和 JSON 之间进行相互转换。
  - 无须考虑用于 XML 和 JSON 相互转换的传统转换工具。
  - 使用在当前平台上运行良好的 XML 类库。
- 编写单元测试来转换 Web API 所提供的 JSON 文档内容。

这些 JSON 转换技术可以将外部 API 所提供的 JSON 数据转换成应用程序所兼容的数据格式。

7.8 内容预告

介绍完 JSON 生态系统（Schema、搜索和转换）后，我们将开始讲述本书最后一部分内容，即 JSON 的企业级应用。这部分内容包括以下话题：

- 超媒体；
- MongoDB（NoSQL）；
- Kafka 消息系统。

在第 8 章中，我们将探讨 JSON 超媒体，并展示其与 API 之间的交互。

第三部分

JSON的企业级应用

第8章

JSON与超媒体

想象一下用 HTML 来搭建一个在浏览器中使用的 Web 应用程序。在该应用程序中，你可以使用标准的 HTML 来添加表单、超链接和按钮，而浏览器无须等待自身版本更新即可渲染这些新增控件。在“远古时代”，事情并不是这样的。那时，如果要发布包含新功能的服务器端应用程序，就不得不同时发布新版客户端代码，使之与服务器端匹配。浏览器的出现改变了这一做法。

如今，我们生活在一个“富客户端”时代，这个时代是以人们手中应用程序的形式呈现出来的。当然，我们可以通过手机来直接访问网页，但出于种种原因，用户和企业都更青睐于拥有图标、可直接在设备上点击打开的原生应用程序。在这种情况下，我们如何才能既拥抱原生应用程序，又保留浏览器式的可配置化特性呢？答案就是超媒体。使用超媒体后，我们不仅可以发送数据，还可以发送用户能够对数据进行的操作，以及触发这些操作的条件。

到目前为止，本书所介绍的 RESTful API 调用及相应的 JSON 响应都是比较独立的，彼此间没有任何关联。演讲者数据 API 返回的所有 JSON 响应都包含了各自对应的演讲者数据，但缺乏与这一数据相关的资源 / 操作的任何信息。

超媒体可以使得 REST API 引导其使用者实现以下功能。

- 链接到其他相关资源（如其他 API）。例如，会议 API 的返回结果中可以包含预订、演讲者或会场 API 的链接，从而让会议 API 的使用者获知更多信息，继而引发购票操作。
- 对 API 返回的数据进行语义化。新增的元数据文档化地描述了 JSON 响应中的数据，并定义了数据元素的含义。
- 对当前返回的 API 资源，定义可以进行的操作。比如，除增删改查外，演讲者数据 API 还可以提供更多操作。例如，提供批量链接以指导演讲者熟悉演讲申请流程，使其能成功地在会议上进行演讲。

超媒体将资源分组打包，引导资源的使用者进行一系列操作并最后实现业务目标。可以将超媒体理解为 Web 上的购物流程在 API 上的等价物：网页可以通过购物车引导消费者完成所有的购物流程并最终付款（幸运的话）。超媒体格式向 API 的使用者提供了一套标准化的解决方案，以便其能解读并处理 API 响应中超链接的数据元素。

本章将对以下知名的 JSON 超媒体格式进行对比：

- Siren
- JSON-LD
- Collection+JSON
- `json:api`
- HAL

8.1 超媒体格式对比

我们将使用之前章节中所提到的演讲者数据来探讨超媒体格式。以下是调用假想的 `myconference` 演讲者数据 API 所返回的结果：

```
GET http://myconference.api.com/speakers/123456

{
  "id": "123456",
  "firstName": "Larson",
  "lastName": "Richard",
  "email": "larson.richard@myconference.com",
  "tags": [
    "JavaScript",
    "AngularJS",
    "Yeoman"
  ],
  "age": 39,
  "registered": true
}
```

如果需要该演讲者所做的一系列报告的信息，那么就发起另一个 API 调用：

```
GET http://myconference.api.com/speakers/123456/presentations

[
  {
    "id": "1123",
    "speakerId": "123456",
    "title": "Enterprise Node",
    "abstract": "Many developers just see Node as a way to build web APIs ...",
    "audience": [
      "Architects",
      "Developers"
    ]
  },
  {
    "id": "2123",
```

```
      "speakerId": "123456",
      "title": "How to Design and Build Great APIs",
      "abstract": "Companies now leverage APIs as part of their online ...",
      "audience": [
        "Managers",
        "Architects",
        "Developers"
      ]
    }
  ]
```

接下来，我们将介绍如何使用多种超媒体格式来表示演讲者和报告信息 API。

8.1.1　定义关键词

在继续深入前，我们先定义与 REST 相关的几个关键词。

资源

保存数据的任何实体都是**资源**，其中包括对象、文档或服务（如股票报价服务）等。一个资源可以关联另一个资源。资源以拥有 URI 终端的形式存在。

表示

资源当前的状态，以 JSON 或 XML 的形式表达。

8.1.2　关于超媒体的个人看法

在评价某项特定技术时，所有的架构师和开发人员都会有自己的看法。在评估和比较所有的超媒体格式前，我会先介绍一下自己对超媒体的看法。超媒体是一项强大的技术，可以在 API 返回的数据中添加丰富的元数据信息，但这项技术同时也充满了争议。很多人热爱这项技术，但也有很多人对此表示厌恶，我的态度则介于两者之间。

REST 和超媒体社区的很多人认为，在 JSON 消息体中添加关于操作和语义定义的元数据是很有用的。我尊重所有人的意见和看法，但只接受出于以下原因而使用外部资源链接的情况。

- 如果一开始就能做好 API 的文档工作，那么有关操作和数据定义的额外信息就毫无必要了。为什么每次 API 调用所返回的 JSON 数据中要包含动作和数据类型信息？这在以下场景中尤其显得混乱。
  - OpenApi（原名 Swagger）、RAML 和 API Blueprint 都可以在 API 文档中提供这一信息。
  - JSON Schema 可以描述 JSON 数据所表示的数据类型。
- 超媒体增加了 API 返回的 JSON 消息体的复杂度。当使用功能丰富的超媒体格式时，不得不考虑以下缺陷。
  - 原始的消息数据被更改了，并且难以解读。本章中介绍的绝大多数超媒体格式都会对原始的消息数据进行修改，这增加了使用者理解和处理 API 的难度。
  - 作为 API 的提供者，你不得不花费更多的时间和精力来解释 API 的具体使用方法，这却无法阻止 API 的使用者选择更简单的替代产品。
  - 消息体过于庞大，会消耗更多的带宽。

- 在消息体中包含其他相关资源的链接是一个很不错的方案，因为在不改变原始 JSON 数据表示的情况下，这种链接方案可以引导 API 的使用者使用整个 API。

8.1.3 Siren

Siren（Structured Interface for Representing Entities，实体表示的结构化接口）是在 2012 年开发的，设计用于表示 Web API 所返回的数据，可与 JSON 和 XML 协同工作。具体可参考其 GitHub 网站。Siren 在 IANA 中的媒体类型为 `application/vnd.siren+json`。

Siren 中的核心概念如下。

实体
: 可以通过 URI 访问的资源称为实体。实体拥有属性和动作。

动作
: 可以在实体上采取的行为。

链接
: 与其他实体之间的链接。

例 8-1 展示了以下 HTTP 请求所返回的 Siren 格式的演讲者数据。

```
GET http://myconference.api.com/speakers/123456
Accept: application/vnd.siren+json
```

例 8-1 data/speaker-siren.json

```
{
  "class": ["speaker"],
  "properties": {
    "id": "123456",
    "firstName": "Larson",
    "lastName": "Richard",
    "email": "larson.richard@myconference.com",
    "tags": [
      "JavaScript",
      "AngularJS",
      "Yeoman"
    ],
    "age": 39,
    "registered": true
  },
  "actions": [
    {
      "name": "add-presentation",
      "title": "Add Presentation",
      "method": "POST",
      "href": "http://myconference.api.com/speakers/123456/presentations",
      "type": "application/x-www-form-urlencoded",
      "fields": [
        {
          "name": "title",
          "type": "text"
```

```
        },
        {
          "name": "abstract",
          "type": "text"
        },
        {
          "name": "audience",
          "type": "text"
        }
      ]
    }
  ],
  "links": [
    { "rel": ["self"],
      "href": "http://myconference.api.com/speakers/123456"
    },
    {
      "rel": ["presentations"],
      "href": "http://myconference.api.com/speakers/123456/presentations"
    }
  ]
}
```

在以上示例中，speaker 实体的定义情况如下。

- class 声明了该资源所属的类（在本例中，这个类为 speaker）。
- properties 是一个保存资源表示的对象。该对象是 API 响应中真正的数据体。
- actions 定义了可以对一个 speaker 采取的动作。在本例中，actions 声明了可以对一个 speaker 添加 presentation。
- links 提供了对 self（当前资源）和 presentation 资源的链接，presentation 资源的 URI 会返回该 speaker 的演讲列表。

在描述实体资源上可采取的动作时，Siren 能够提供非常好的元数据。Siren 中也有用于描述数据的类（类型），但没有像 JSON-LD 这样的数据定义（语义声明）。

8.1.4 JSON-LD

JSON-LD（JavaScript Object Notation for Linking Data，链接数据的 JavaScript 对象表示法）于 2014 年成为 W3C 标准。JSON-LD 是一种设计用于 REST API 的数据链接格式，可与 MongoDB、CouchDB 等 NoSQL 数据库一同使用。如需了解更多信息，可参考其官方网站和 GitHub 页面。JSON-LD 的媒体类型是 application/ld+json，文件扩展名为 .jsonld。因为是 W3C 标准，所以 JSON-LD 拥有活跃的社区和庞大的工作组。

例 8-2 展示了以下 HTTP 请求返回的 JSON-LD 格式的演讲者数据。

```
GET http://myconference.api.com/speakers/123456
Accept: application/vnd.ld+json
```

例 8-2　data/speaker.jsonld

```
{
  "@context": {
```

```
      "@vocab": "http://schema.org/Person",
      "firstName": "givenName",
      "lastName": "familyName",
      "email": "email",
      "tags": "http://myconference.schema.com/Speaker/tags",
      "age": "age",
      "registered": "http://myconference.schema.com/Speaker/registered"
    },
    "@id": "http://myconference.api.com/speakers/123456",
    "id": "123456",
    "firstName": "Larson",
    "lastName": "Richard",
    "email": "larson.richard@myconference.com",
    "tags": [
      "JavaScript",
      "AngularJS",
      "Yeoman"
    ],
    "age": 39,
    "registered": true,
    "presentations": "http://myconference.api.com/speakers/123456/presentations"
  }
```

在以上示例中，`@context` 对象提供了演讲者数据的总体上下文信息。具体来说，除了列举出数据字段，`@context` 还与 `@vocab` 一起，对组成 `speaker` 对象的数据元素进行了精确的语义定义。以下是一些具体解释。

- Schema.org 网站提供了对年龄和个体等常用数据元素的精确定义。
- `@vocab` 将基本类型设置为个体，然后允许使用其他字段（如 `tags` 或 `registered`）来扩展该类型，以供 `speaker` 使用。
- `@id` 本质上是一个 URI，表示访问某个特定 `speaker` 所用的 ID。

值得注意的是，用于表示 `speaker` 的核心 JSON 并未有所改变，而这也是 JSON-LD 在有遗留 API 时的一大卖点。这种增量式改进方案可以避免破坏已有的 API 使用，进而更容易逐渐采纳 JSON-LD。在这一方案中，已有的 JSON 可以保持原样，开发者则可迭代地在 API 数据中添加数据链接的语义信息。

另外，http://myconference.schema.com 这一网站其实并不存在。事实上，示例中使用该网站仅仅只是为了演示而已。如果需要使用 Schema.org 上某个不存在的定义，那么只要能够确保提供良好的文档，就可以直接在自己的域名下创建相关定义。

例 8-3 展示了以下 HTTP 请求所返回的 JSON-LD 格式的演讲者演说列表数据。

```
GET http://myconference.api.com/speakers/123456/presentations
Accept: application/vnd.ld+json
```

例 8-3　data/presentations.jsonld

```
{
  "@context": {
    "@vocab": "http://myconference.schema.com/",
    "presentations": {
      "@type": "@id",
```

```
        "id": "id",
        "speakerId": "speakerId",
        "title": "title",
        "abstract": "abstract",
        "audience": "audience"
      }
    },
    "presentations": [
      {
        "@id": "http://myconference.api.com/speakers/123456/presentations/1123",
        "id": "1123",
        "speakerId": "123456",
        "title": "Enterprise Node",
        "abstract": "Many developers just see Node as a way to build web APIs or ...",
        "audience": [
          "Architects",
          "Developers"
        ]
      }, {
        "@id": "http://myconference.api.com/speakers/123456/presentations/2123",
        "id": "2123",
        "speakerId": "123456",
        "title": "How to Design and Build Great APIs",
        "abstract": "Companies now leverage APIs as part of their online strategy ...",
        "audience": [
          "Managers",
          "Architects",
          "Developers"
        ]
      }
    ]
  }
```

在以上示例中，`@context` 表示所有的数据都与 `presentations` 概念有关。因为 `http://myconference.schema.com/presentations` 对象不存在，所以我们需要显式声明 `presentations`。如果 `http://myconference.schema.com/presentations` 对象真的存在，则 `@context` 如下所示：

```
"@context": "http://myconference.schema.com/presentations"
```

你也可以在 JSON-LD Playground 上测试一下前面的示例。JSON-LD Playground 是一个用于校验 JSON-LD 文档的非常不错的在线测试工具。在编写 API 代码前，可以使用该工具来校验数据格式。

JSON-LD 本身既不提供有关操作的信息，也不提供有关数据表示的语义。作为 JSON-LD 的插件，HYDRA 可以为客户端—服务器端通信提供更多的描述词汇。

如需了解更多有关 HYDRA 的信息，可参考以下资源：

- HYDRA 官方网站；
- W3C 社区。

例 8-4 展示了增加 HYDRA 操作的 JSON-LD 格式的演讲者演说列表数据。

```
GET http://myconference.api.com/speakers/123456/presentations
Accept: application/vnd.ld+json
```

例 8-4　data/presentations-operations.jsonld

```
{
  "@context": [
    "http://www.w3.org/ns/hydra/core", {
      "@vocab": "http://myconference.schema.com/",
      "presentations": {
        "@type": "@id",
        "id": "id",
        "speakerId": "speakerId",
        "title": "title",
        "abstract": "abstract",
        "audience": "audience"
      }
    }
  ],
  "presentations": [
    {
      "@id": "http://myconference.api.com/speakers/123456/presentations/1123",
      "id": "1123",
      "speakerId": "123456",
      "title": "Enterprise Node",
      "abstract": "Many developers just see Node as a way to build web APIs or ...",
      "audience": [
        "Architects",
        "Developers"
      ]
    }, {
      "@id": "http://myconference.api.com/speakers/123456/presentations/2123",
      "id": "2123",
      "speakerId": "123456",
      "title": "How to Design and Build Great APIs",
      "abstract": "Companies now leverage APIs as part of their online strategy ...",
      "audience": [
        "Managers",
        "Architects",
        "Developers"
      ]
    }
  ],
  "operation": {
    "@type": "AddPresentation",
    "method": "POST",
    "expects": {
      "@id": "http://schema.org/id",
      "supportedProperty": [
        {
          "property": "title",
          "range": "Text"
        }, {
          "property": "abstract",
          "range": "Text"
        }
      ]
    }
  }
}
```

在以上示例中：

- `operation` 声明可以通过 `POST` 请求增加演说实体；
- `@context` 引用了 HYDRA 域，从而在格式中添加 `operation` 关键词；
- `@vocab` 添加了 http://myconference.schema.com/ 域和 `presentations` 定义。

由于可以在不改变原始数据的情况下添加其他相关资源的链接，JSON-LD 本身就是一个非常优秀的工具。换而言之，对 API 的使用者来说，使用 JSON-LD 不会产生任何破坏性的变更。出于简洁性的考虑，不建议使用 JSON-LD 时引入 HYDRA。

8.1.5 Collection+JSON

Collection+JSON 创建于 2011 年，专注于处理集合中的数据成员，类似于 Atom 这种订阅 / 供稿格式。如需了解更多相关信息，可参见 Collection+JSON 的官方网站和 GitHub 主页。Collection+JSON 的媒体类型为 `application/vnd.collection+json`。

合法的 Collection+JSON 响应必须拥有一个名为 `collection` 的根字段，该字段值为包含以下内容的对象。

- 版本号（`version`）。
- 值为 URI 的 `href`，该 URI 指向 `self` 资源（请求的原始资源）。

例 8-5 展示了以下 HTTP 请求返回的 Collection+JSON 格式的演讲者数据。

```
GET http://myconference.api.com/speakers/123456
Accept: application/vnd.collection+json
```

例 8-5 data/speaker-collection-json-links.json

```
{
  "collection": {
    "version": "1.0",
    "href": "http://myconference.api.com/speakers",
    "items": [
      {
        "href": "http://myconference.api.com/speakers/123456",
        "data": [
          { "name": "id", "value": "123456" },
          { "name": "firstName", "value": "Larson" },
          { "name": "lastName", "value": "Richard" },
          { "name": "email", "value": "larson.richard@myconference.com" },
          { "name": "age", "value": "39" },
          { "name": "registered", "value": "true" }
        ],
        "links": [
          {
            "rel": "presentations",
            "href": "http://myconference.api.com/speakers/123456/presentations",
            "prompt": "presentations"
          }
        ]
      }
    ]
  }
}
```

对于以上示例，需要注意以下几点。

- `collection` 对象封装了演讲者数据。
- `items` 数组包含了演讲者集合中的所有对象。因为请求中包含 ID，所以该集合中只有一个对象。
- `data` 数组包含了组成一个演讲者的所有数据元素的名称 – 值对。
- `links` 数组提供了演讲者相关资源的链接。每个链接包含以下内容。
  - `rel` 关键词对资源关系进行描述。
  - `href` 提供了当前演讲者的 `presentations` 资源的超链接。
  - HTML 表单可以使用 `prompt` 来引用演讲者集合。

Collection+JSON 还支持读取、写入和查询集合中的成员，但对 Collection+JSON 的完整讨论超出了本书的探讨范围。你可以参考 http://amundsen.com/media-types/collection/examples/ 来获取更多示例，也可以参考 http://amundsen.com/media-types/tutorials/collection/tutorial-01.html 来学习相关教程。

Collection+JSON 在提供资源的链接关系方面做了非常不错的工作，但通过将数据转换为 `data` 数组中的名称 – 值对，这一格式彻底改变了演讲者数据的结构。

8.1.6 json:api

`json:api` 于 2013 年开发，用于对 API 中 JSON 请求 / 响应的格式进行标准化约定。虽然 `json:api` 的主要关注点在 API 请求 / 响应的数据上，但其实它也包含了超媒体的部分内容。可在其官方网站和 GitHub 主页上了解更多信息。`json:api` 的媒体类型为 `application/vnd.api+json`。

一个合法的 `json:api` 文档**必须**拥有以下元素之一作为根字段。

`data`
: 资源的数据表示，其中包含一个或多个资源对象，每个资源对象都必须拥有 `type` 字段（表示数据类型）和 `id` 字段（表示资源的唯一标识 ID）。

`errors`
: 错误对象的数组，每个错误对象包含调用 API 时所得到的错误码及错误消息。

`meta`
: 包含非标准的元数据信息（如版权、作者等）。

可选的根元素如下。

`links`
: 保存相关资源超链接的对象。

`included`
: 保存相关内置资源对象的数组。

例 8-6 展示了以下 HTTP 请求所返回的 `json:api` 格式的演讲者数据列表。

```
GET http://myconference.api.com/speakers
Accept: application/vnd.api+json
```

例 8-6　data/speakers-jsonapi-links.json


```
{
  "links": {
    "self": "http://myconference.api.com/speakers",
    "next": "http://myconference.api.com/speakers?limit=25&offset=25"
  },
  "data": [
    {
      "type": "speakers",
      "id": "123456",
      "attributes": {
        "firstName": "Larson",
        "lastName": "Richard",
        "email": "larson.richard@myconference.com",
        "tags": [
          "JavaScript",
          "AngularJS",
          "Yeoman"
        ],
        "age": 39,
        "registered": true
      }
    },
    {
      "type": "speakers",
      "id": "223456",
      "attributes": {
        "firstName": "Ester",
        "lastName": "Clements",
        "email": "ester.clements@myconference.com",
        "tags": [
          "REST",
          "Ruby on Rails",
          "APIs"
        ],
        "age": 29,
        "registered": true
      }
    },
    ...
  ]
}
```


以上示例的工作机制如下。

- `links` 数组提供了与当前演讲者数据相关的资源的链接。在本例中，每个元素都包含了相关资源的 URI。注意，链接的命名没有任何限制，但一般使用 `self` 来表示当前资源，`next` 则往往用于分页功能。
- `data` 数组中包含了资源对象的列表，其中每个对象都拥有 `type` 字段（如 `speakers`）和 `id` 字段，以满足 `json:api` 格式的要求。`attributes` 对象中保存了组成每个 `speaker` 对象的名称 – 值对。

例 8-7 展示了如何使用 `json:api` 来嵌入有关 `speaker` 资源的所有 `presentation` 对象。

```
GET http://myconference.api.com/speakers/123456
Accept: application/vnd.api+json
```

例 8-7　data/speaker-jsonapi-embed-presentations.json


```
{
  "links": {
    "self": "http://myconference.api.com/speakers/123456"
  },
  "data": [
    {
      "type": "speaker",
      "id": "123456",
      "attributes": {
        "firstName": "Larson",
        "lastName": "Richard",
        "email": "larson.richard@myconference.com",
        "tags": [
          "JavaScript",
          "AngularJS",
          "Yeoman"
        ],
        "age": 39,
        "registered": true
      }
    }
  ],
  "included": [
    {
      "type": "presentations",
      "id": "1123",
      "speakerId": "123456",
      "title": "Enterprise Node",
      "abstract": "Many developers just see Node as a way to build web APIs or ...",
      "audience": [
        "Architects",
        "Developers"
      ]
    }, {
      "type": "presentations",
      "id": "2123",
      "speakerId": "123456",
      "title": "How to Design and Build Great APIs",
      "abstract": "Companies now leverage APIs as part of their online ...",
      "audience": [
        "Managers",
        "Architects",
        "Developers"
      ]
    }
  ]
}
```

在以上示例中，included 数组（json:api 标准的一部分）声明了内嵌的 presentations 资源。虽然可以减少 API 的调用次数，但这种内嵌资源的方式造成了资源间的数据紧耦合，因为 speaker 资源必须知道 presentation 数据的格式和内容。

例 8-8 通过 links 提供了一种更好的方式来展现资源间的关系。

```
GET http://myconference.api.com/speakers/123456
Accept: application/vnd.api+json
```

例 8-8 data/speaker-jsonapi-link-presentations.json

```
{
  "links": {
    "self": "http://myconference.api.com/speakers/123456",
    "presentations": "http://myconference.api.com/speakers/123456/presentations"
  },
  "data": [
    {
      "type": "speaker",
      "id": "123456",
      "attributes": {
        "firstName": "Larson",
        "lastName": "Richard",
        "email": "larson.richard@myconference.com",
        "tags": [
          "JavaScript",
          "AngularJS",
          "Yeoman"
        ],
        "age": 39,
        "registered": true
      }
    }
  ]
}
```

在以上示例中，links 数组显示 speaker 拥有 presentations 这一资源，同时也提供了相应的 URI；但这一次 speaker 资源（以及相关 API）并不知道 presentation 资源中的具体数据。另外，在此方案下，API 的使用者需要处理的数据也更少。这一松耦合的设计使得 presentation 数据可以在不影响演讲者 API 的情况下进行变更。

json:api 中包含了丰富的特性，具体包括标准化的错误消息、分页、内容协商，以及创建 / 更新 / 删除资源的策略等。我曾借用 json:api 标准中的部分内容来创建过 API 规范。同时，绝大多数编程平台都能提供优秀的类库来简化 json:api 的使用。data 数组及其内部资源对象（要求具备 type 和 id 字段）会更改原始的 JSON 数据表示，其他对象内容则保持不变。对 json:api 的完整讨论超出了本书的探讨范围，如需了解更多相关信息，可访问其示例网页和完整的标准文档。

8.1.7 HAL

HAL（Hypertext Application Language，超文本应用程序语言）于 2012 年成为 IETF 标

准，设计用于以超链接的形式来关联资源，可与 JSON 和 XML 协同工作。如需了解更多信息，可参考其官方网站和 GitHub 主页。HAL 的媒体类型为 application/hal+json 和 application/hal+xml。

HAL 的格式简单、可读性强，也不会改变原始的数据表示。作为一种流行的媒体类型，HAL 基于以下两个概念而设计。

资源对象

资源包含了（保存在 _links 对象中的）链接、其他资源，以及保存在 _embedded 对象中的内嵌资源（如订单资源中包含了商品资源）。

链接

链接提供了其他外部资源的 URI。

虽然 _embedded 和 _links 对象都是可选的，但其中之一必须出现在合法 HAL 文档的对象根字段中。

例 8-9 展示了以下 HTTP 请求返回的 HAL 格式的演讲者数据。

```
GET http://myconference.api.com/speakers/123456
Accept: application/vnd.hal+json
```

例 8-9　data/speaker-hal.json

```
{
  "_links": {
    "self": {
      "href": "http://myconference.api.com/speakers/123456"
    },
    "presentations": {
      "href": "http://myconference.api.com/speakers/123456/presentations"
    }
  },
  "id": "123456",
  "firstName": "Larson",
  "lastName": "Richard",
  "email": "larson.richard@myconference.com",
  "tags": [
    "JavaScript",
    "AngularJS",
    "Yeoman"
  ],
  "age": 39,
  "registered": true
}
```

以上示例的工作机制如下。

- _links 对象包含了链接关系信息，其中每条信息都显示了链接的具体语义。
  - href 字段在链接关系中是必备的。href 的值必须是一个合法的 URI（参考 RFC 3986）或 URI 模板（参考 RFC 6570）。
- 有关链接关系的具体描述如下。

- self 为当前 speaker 资源的链接。
- presentations 为当前 speaker 将要演说的资源。在本例中，presentations 对象通过 href 描述当前资源和 http://myconference.api.com/speakers/123456/presentations 超链接之间的关系。
- 注意，next 和 find **不是** HAL 的关键词。HAL 允许使用自定义名称来描述链接对象。

接下来我们获取演讲者列表，从而使得以上示例变得更有意思，如例 8-10 所示。

```
GET http://myconference.api.com/speakers
Accept: application/vnd.hal+json
```

例 8-10　data/speakers-hal-links.json

```
{
  "_links": {
    "self": {
      "href": "http://myconference.api.com/speakers"
    },
    "next": {
      "href": "http://myconference.api.com/speakers?limit=25&offset=25"
    },
    "find": {
      "href": "http://myconference.api.com/speakers{?id}", "templated": true
    }
  },
  "speakers": [
    {
      "id": "123456",
      "firstName": "Larson",
      "lastName": "Richard",
      "email": "larson.richard@myconference.com",
      "tags": [
        "JavaScript",
        "AngularJS",
        "Yeoman"
      ],
      "age": 39,
      "registered": true
    },
    {
      "id": "223456",
      "firstName": "Ester",
      "lastName": "Clements",
      "email": "ester.clements@myconference.com",
      "tags": [
        "REST",
        "Ruby on Rails",
        "APIs"
      ],
      "age": 29,
      "registered": true
    },
    ...
  ]
}
```

以上示例的工作机制如下。

- 除了 `self`，还包括以下链接关系。
  - `next` 声明了接下来的一批 `speaker` 资源。换而言之，这是 API 提供分页功能的一种方式。在本例中，`limit` 参数表示每次 API 调用都会返回 25 个 `speaker` 对象，`offset` 参数则表示列表中首个对象为第 26 个 `speaker`。这一约定与 Facebook 的分页风格比较像。
  - `find` 以模板链接的形式提供用于搜索单个 `speaker` 的超链接。模板中的 `{?id}` 表示可以在 URI 中通过 `id` 来搜索 `speaker`，`templated` 属性则表示当前链接是一个模板链接。
- 原始的 JSON 数据表示保持不变。

回到第一个示例，我们将所有的 `presentation` 对象嵌入 `speaker` 资源，如例 8-11 所示。

```
GET http://myconference.api.com/speakers/123456
Accept: application/vnd.hal+json
```

例 8-11　/data/speaker-hal-embed-presentations.json

```
{
  "_links": {
    "self": {
      "href": "http://myconference.api.com/speakers/123456"
    },
    "presentations": {
      "href": "http://myconference.api.com/speakers/123456/presentations"
    }
  },
  "_embedded": {
    "presentations": [
      {
        "_links": {
          "self": {
            "href": "http://myconference.api.com/speakers/123456/presentations/1123"
          }
        },
        "id": "1123",
        "title": "Enterprise Node",
        "abstract": "Many developers just see Node as a way to build web APIs ...",
        "audience": [
          "Architects",
          "Developers"
        ]
      },
      {
        "_links": {
          "self": {
            "href": "http://myconference.api.com/speakers/123456/presentations/2123"
          }
        },
        "id": "2123",
```

```
          "title": "How to Design and Build Great APIs",
          "abstract": "Companies now leverage APIs as part of their online ...",
          "audience": [
            "Managers",
            "Architects",
            "Developers"
          ]
        }
      ]
    },
    "id": "123456",
    "firstName": "Larson",
    "lastName": "Richard",
    "email": "larson.richard@myconference.com",
    "tags": [
      "JavaScript",
      "AngularJS",
      "Yeoman"
    ],
    "age": 39,
    "registered": true
  }
```

在以上示例中，相较于使用 presentations 链接关系，我们使用了 _embedded 对象将 presentation 对象内嵌到 speaker 中。每个 presentation 对象都拥有包含相关资源的 _links 子对象。

乍一看，内嵌相关资源的这种做法还挺合理，但出于以下原因，我更喜欢使用之前的资源链接的方案。

- 内嵌资源的方案增加了消息体的规模。
- _embedded 对象更改了数据表示。
- 内嵌方案耦合了演讲者 API 和演说 API。使用该方案后，演讲者 API 必须知道演说资源的数据结构。如果使用资源链接方案,那么演讲者 API 只需要知道相关 API 的存在即可。

刨除内嵌资源方案，HAL 是一种轻量级的格式，可以在不更改原始的数据表示的情况下提供其他资源的链接。

8.2 结论

以下是有关超媒体的总结：保持简单。保持原始资源数据的结构不变。如果将提供稳定、可靠的 API 文档作为设计流程的一个步骤，那么这一步骤本身就已经满足了很大一部分对超媒体技术需求的初衷。对我而言，超媒体最有用的部分在于和其他资源的链接。“彻底超媒体化”的支持者可能会强烈反对我的观点（没关系），以下则是我的反驳。

- 用户不会使用难以理解的 API。
- 原始的 JSON 数据是最重要的。不能仅仅为了遵循超媒体格式就更改资源的结构。

基于这些考虑，我选择最小化的 HAL 结构（仅包含链接，不包含内嵌资源）作为超媒体

格式。即使存在以上顾虑，HAL 依旧是一种优秀的格式，因为它：

- 是可以工作的最简单方案；
- 是一项标准；
- 拥有广泛的社区支持；
- 拥有可靠的跨平台类库；
- 不改变原始的 JSON 数据表示；
- 对数据的语义和操作没有要求；
- 只做必需的工作，不会画蛇添足。

`json:api` 是我的第二选择（使用资源链接而非内嵌资源），原因在于：除提供超媒体功能外，`json:api` 还可以对 JSON 请求 / 响应进行标准化，同时依旧保持原始 JSON 数据的完整性与设计意图。在对 JSON 数据有所更改的超媒体格式中，`json:api` 算是影响最小的。因为 `json:api` 具备广泛的跨平台支持，所以你可以使用编程类库来减少格式化所需的工作，缩短并简化开发过程。如果在超媒体格式外还需要对整个企业内所有 API 中的 JSON 请求 / 响应进行标准化（有关 API 设计的话题超出了本书的探讨范围），那么 `json:api` 方案是值得着重考虑的。

（刨除 HYDRA 的）JSON-LD 是我的第三选择，因为其简洁，而且不更改原始的 JSON 数据表示。虽然在已有的 API 上添加数据语义并不难，但我认为没有必要这么做：在定义数据的语义与结构这一点上，高质量的 API 文档 +JSON Schema 这样的方案会更好一些。

8.3 建议

你可能不赞同我对超媒体的看法，但想象一下，如果你作为一名架构师或团队负责人向整个团队发出“在开发 API 时彻底使用超媒体的全部特性”这样的指令，那么团队中的开发人员会将超媒体视作有益的技术补充，还是累赘？回想一下刚提出极限编程思想的那些岁月吧，采用可以工作的最简方案。使用正确的工具和技术来完成工作，同时建议采取以下策略。

- 使用 OpenApi/Swagger 或 RAML 对 API 进行文档化。
- 使用 JSON Schema 来定义数据结构。
- 选择 HAL、`json:api` 或 JSON-LD 作为超媒体格式，并从关联其他资源的简单链接开始做起。
- 如何评估开发流程的状态。
  - 团队的开发速度如何？
  - API 的可测试性如何？
- 向 API 的使用者寻求反馈。反馈内容如下。
  - 数据表示是否容易理解？
  - 数据是否容易读取和使用？
- 尽早进行快速迭代和评估。

使用上述策略后再分析是否有必要在 JSON 中添加操作和数据定义；答案很可能是否定的。

8.4 实际中遇到的问题

如果考虑在 API 中使用超媒体技术，则最好事先思考以下几个问题。

- 超媒体在社区中的理解度不高。当我就相关话题进行分享时，很多开发者根本就没听说过超媒体，或者知之甚少，而且对超媒体的具体作用也是茫然无知。即使是最简单的超媒体格式，也需要一些社区教育工作来普及。
- 标准缺失。超媒体格式有很多种，本章介绍了 5 种最主流的，其中只有 2 种（HAL 和 JSON-LD）是拥有相关标准的。由此可见，技术社区在超媒体上的意见并不一致。
- 对于 API 的提供者和使用者来说，无论何种格式，超媒体的使用都会引入额外的序列化 / 反序列化工作。因此，需要确保采用广为接受的超媒体格式，以获得跨平台的类库支持。这么做可以使得开发人员的工作变得更加简单。在后文使用 HAL 进行测试时，我们将介绍相关内容。

8.5 在演讲者数据API中用HAL进行测试

与之前的章节相同，我们将在不编写任何代码的情况下，对提供 JSON 响应的模拟 API 进行测试。

8.5.1 测试数据

我们将使用之前章节中所提到的演讲者数据作为测试数据（可在 GitHub 上找到相关信息），并将其部署为 RESTful API，以创建模拟的测试 API。同样，我们依旧使用 `json-server` 这一 Node.js 模块将 speakers.json 文件暴露为 Web API。如需安装 `json-server`，可参考 A.2.5 节中的内容。

可在本机中以 5000 端口运行 `json-server`：

```
cd chapter-8/data

json-server -p 5000 ./speakers-hal-server-next-rel.json
```

在 Postman 中访问 http://localhost:5000/speakers，选择 `GET` 作为 HTTP 方法，然后点击 Send 按钮。应该可以观察到模拟 API 所提供的所有演讲者信息，如图 8-1 所示。

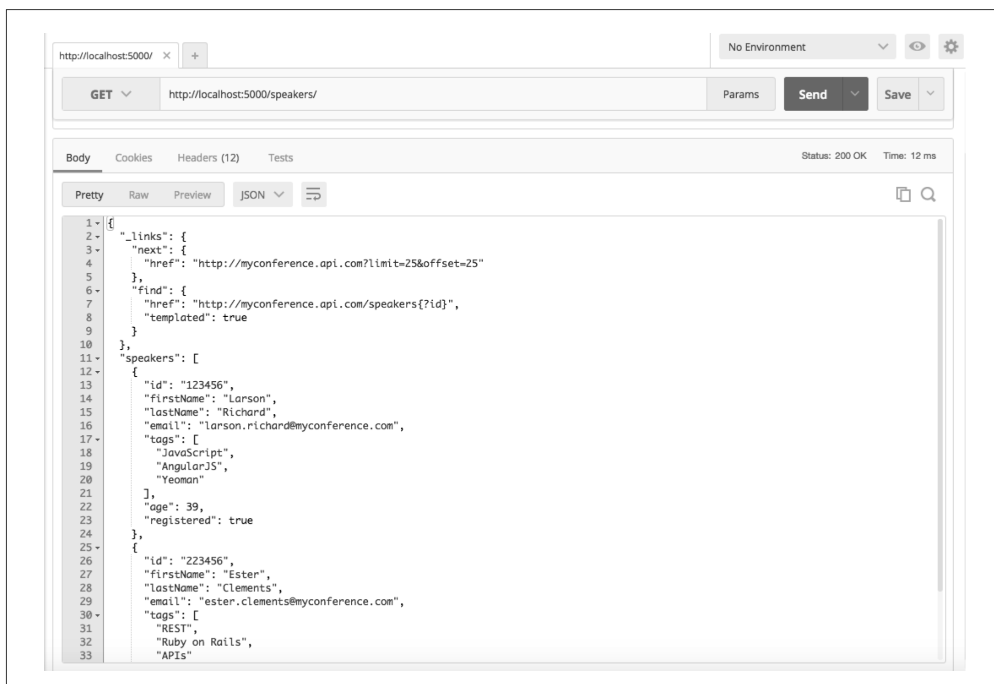


图 8-1：在 Postman 中访问由 `json-server` 所提供的 HAL 格式的演讲者数据

也可以通过浏览器访问该 URI。

值得注意的是，在这一示例中，我们必须向 `json-server` 传递演讲者数据文件信息，才能成功启动服务器。例 8-12 展示了更新后的 HAL 格式结构。

例 8-12　data/speakers-hal-server-next-rel.json

```
{
  "speakers": {
    "_links": {
      "self": {
        "href": "http://myconference.api.com/speakers"
      },
      "next": {
        "href": "http://myconference.api.com?limit=25&offset=25"
      },
      "find": {
        "href": "http://myconference.api.com/speakers{?id}",
        "templated": true
      }
    },
    "speakers": [{
      "id": "123456",
      "firstName": "Larson",
      "lastName": "Richard",
      "email": "larson.richard@myconference.com",
      "tags": [
        "JavaScript",
```

```
        "AngularJS",
        "Yeoman"
      ],
      "age": 39,
      "registered": true
    }, {
      "id": "223456",
      "firstName": "Ester",
      "lastName": "Clements",
      "email": "ester.clements@myconference.com",
      "tags": [
        "REST",
        "Ruby on Rails",
        "APIs"
      ],
      "age": 29,
      "registered": true
    }]
  }
}
```

在该示例中，最外面的 speakers 对象结构是必需的，它使得 json-server 根据 URI（http://localhost:5000/speakers）自动提供相应的文件；剩余的数据结构（links 对象和 speakers 数组）则保持不变。

8.5.2 HAL单元测试

准备好 API 后，我们将介绍单元测试的创建。与之前的章节相同，我们将继续使用 Node.js 中的 Mocha/Chai。继续阅读前，确保已经成功设置好了测试环境。如尚未安装 Node.js，可参考 A.2 节和 A.2.5 节中的内容。如需依照本节的描述运行代码示例中的项目，可使用 cd 命令切换到 chapter-8/myconference 目录，并执行以下命令来安装项目依赖：

```
npm install
```

如需手动创建本节中的 Node.js 项目，可参考本书在 GitHub 上的相关指导步骤。

以下是本节单元测试中使用到的 npm 模块。

Unirest
: 我们在之前的章节中使用该模块来发起 RESTful API 调用。

halfred
: 可从 https://www.npmjs.com/package/halfred 下载的 HAL 解析器，其 GitHub 主页为 https://github.com/traverson/halfred。

例 8-13 展示了如何校验模拟演讲者数据 API 所提供的 HAL 响应。

例 8-13 speakers-hal-test/test/hal-spec.js

```
'use strict';

var expect = require('chai').expect;
var unirest = require('unirest');
var halfred = require('halfred');
```

```
describe('speakers-hal', function() {
  var req;

  beforeEach(function() {
    halfred.enableValidation();
    req = unirest.get('http://localhost:5000/speakers')
      .header('Accept', 'application/json');
  });

  it('should return a 200 response', function(done) {
    req.end(function(res) {
      expect(res.statusCode).to.eql(200);
      expect(res.headers['content-type']).to.eql(
        'application/json; charset=utf-8');
      done();
    });
  });

  it('should return a valid HAL response validated by halfred', function(
    done) {
    req.end(function(res) {
      var speakersHALResponse = res.body;

      var resource = halfred.parse(speakersHALResponse);
      var speakers = resource.speakers;
      var speaker1 = null;

      console.log('\nValidation Issues: ');
      console.log(resource.validationIssues());
      expect(resource.validationIssues()).to.be.empty;
      console.log(resource);
      expect(speakers).to.not.be.null;
      expect(speakers).to.not.be.empty;
      speaker1 = speakers[0];
      expect(speaker1.firstName).to.not.be.null;
      expect(speaker1.firstName).to.eql('Larson');
      done();
    });
  });
});
```

该单元测试的运行机制如下。

- `beforeEach()` 函数在每个单元测试用例运行前执行一次，并完成以下操作。
  - 调用 `halfred.enableValidation()` 来配置 `halfred` 类库，以启用 HAL 校验功能。
  - 调用 http://localhost:5000/speakers 这一 URI 上的模拟 API。
- `'should return a 200 response'` 这一测试用例确保了模拟 API 可以成功返回 HTTP 响应。
- `'should return a valid HAL response validated by halfred'` 这一测试用例中包含了主要的测试工作，具体如下。
  - 调用 `halfred.parse()` 来解析模拟 API 所返回的 HAL 响应。该调用返回一个包含 HAL 链接和剩余 JSON 消息体的 `halfred Response` 对象。如需了解更多信息，可参考 `halfred` 文档。
  - 调用 `resource.validationIssues()` 来校验 HAL 响应，并使用 `chai` 检查校验结果。在接下来使用非法数据运行的单元测试中，我们将再次看到这一调用的实际使用情况。

– 使用 chai 确保 halfred 的 Response 对象中依旧包含了原始的 speakers 数组信息。

使用 npm test 来运行单元测试，因为模拟 API 所提供的数据是合法的，所以单元测试会成功通过。运行后可以观察到以下结果：

```
> mocha test



  speakers-hal
    ✓ should return a 200 response

Validation Issues:
[]
Resource {
  _links: { self: [ [Object] ], next: [ [Object] ], find: [ [Object] ] },
  _curiesMap: {},
  _curies: [],
  _resolvedCuriesMap: {},
  _embedded: {},
  _validation: [],
  speakers:
   [ { id: '123456',
       firstName: 'Larson',
       lastName: 'Richard',
       email: 'larson.richard@myconference.com',
       tags: [Object],
       age: 39,
       registered: true },
     { id: '223456',
       firstName: 'Ester',
       lastName: 'Clements',
       email: 'ester.clements@myconference.com',
       tags: [Object],
       age: 29,
       registered: true } ],
  _original:
   { _links: { self: [Object], next: [Object], find: [Object] },
     speakers: [ [Object], [Object] ] } }
    ✓ should return a valid HAL response validated by halfred


  2 passing (62ms)
```

介绍了如何校验 HAL 数据后，我们将修改由模拟 API 所提供的数据，使其返回非法的结果。我们在 _links 对象中删除了 self 链接，如例 8-14 所示。

例 8-14 data/speakers-hal-server-next-rel-invalid.json

```
{
  "speakers": {
    "_links": {
      "next": {
        "href": "http://myconference.api.com?limit=25&offset=25"
      },
      "find": {
        "href": "http://myconference.api.com/speakers{?id}",
        "templated": true
      }
    },
    ...
  }
}
```

你可能还记得，HAL 标准要求 _links 对象中必须包含 self 引用。使用以下命令重启 json-server，使其提供非法的 HAL 数据：

```
cd chapter-8/data

json-server -p 5000 ./speakers-hal-server-next-rel-invalid.json
```

重新运行单元测试，可以看到 halfred 捕获了 HAL 校验中出现的问题，且该测试也以失败告终：

```
> mocha test



  speakers-hal
    ✓ should return a 200 response

Validation Issues:
[ { path: '$._links',
    message: 'Resource does not have a self link' } ]
    1) should return a valid HAL response validated by halfred


  1 passing (64ms)
  1 failing

  1) speakers-hal should return a valid HAL response validated by halfred:
     Uncaught AssertionError: expected [ Array(1) ] to be empty
      at test/hal-spec.js:36:48
      at Request.handleRequestResponse [as _callback] (node_modules/unirest/index.js:463:26)
      at Request.self.callback (node_modules/request/request.js:187:22)
      at Request.<anonymous> (node_modules/request/request.js:1044:10)
      at IncomingMessage.<anonymous> (node_modules/request/request.js:965:12)
      at endReadableNT (_stream_readable.js:905:12)



npm ERR! Test failed.  See above for more details.
```

8.6 在服务器端使用HAL

至此，我们通过单元测试介绍了如何在客户端中使用 HAL，与此同时，服务器端部署的却是模拟数据（使用了 json-server 和遵循 HAL 标准的静态 JSON 文件）。为了专注于介绍 JSON，本书在服务器端的内容方面着墨不多。以下是使得 RESTful API 返回 HAL 响应的一些服务器端类库。

Java

Spring 的 HATEOS 对 Java 中基于 Spring 的 RESTful API 提供了 HAL 支持。可以在 Spring 文档中找到高质量的教程。

Ruby on Rails

roar gem 在 Ruby on Rails 中提供 HAL 支持。

JavaScript/NodeJS

express-hal 对基于 Express 的 NodeJS RESTful API 提供 HAL 支持。

无论开发平台是哪个、所选的超媒体格式是哪种，在将其选定为最终方案前，需要确保进行一次快速实现以测试该类库。确保所使用的类库易用性强且不会对已有产品造成阻碍是非常重要的。

8.7 深入学习超媒体

本章只是介绍了超媒体技术的一些皮毛。如需深入学习，可参考以下资源：

- Leonard Richardson 所著的《RESTful Web APIs 中文版》；
- Jim Webber 所著的 *REST in practice: Hypermedia and Systems Architecture*（O'Reilly 出版社）。

8.8 本章回顾

本章通过以下内容介绍了如何在 JSON 中使用超媒体：

- 比较一些知名的 JSON 超媒体格式；
- 讨论在 API 中添加超媒体时的顾虑；
- 使用 HAL 来支持演讲者数据 API 的测试。

8.9 内容预告

介绍了如何在 JSON 中使用超媒体后，第 9 章将展示 JSON 在 MongoDB 中的使用情况。

第 9 章

JSON与MongoDB

MongoDB 是一种 NoSQL 数据库，允许开发人员以文档的形式存储数据。基于文档的这一方案能够与同样基于文档设计的 JSON 完美协作。MongoDB 中的数据模型是分层结构的，对我们在 JSON 文档中所看到的丰富的数据类型有着良好的支持。与 JSON 文档一样，由于文档和对象相互兼容，MongoDB 中的文档可以很好地与面向对象编程平台进行整合，开发者也无须使用太多映射逻辑即可便捷地读写数据库。这一方案符合开发者的直觉，同时也减轻了访问数据库所需的开发工作。

本章将介绍以下操作：

- 将 JSON 文档导入 MongoDB；
- 在 MongoDB 中执行主要的几个增删改查操作；
- 将 MongoDB 中的数据导出为 JSON 文档；
- 在不编写代码的情况下，以模拟 RESTful API 的形式访问 MongoDB。

本章专注于介绍 JSON 在 MongoDB 中的使用情况，并适当阐述数据库操作。本章不涉及使用 MongoDB 来开发应用程序的相关内容，这些内容足以写成另外一本书了。如需全面了解 MongoDB 所提供的丰富功能，推荐阅读 Kyle Banker 所著的《MongoDB 实战（第二版)》。

9.1　关于BSON

你可能在 MongoDB 的文档中看到过二进制 JSON（Binary JSON，BSON）这样的词汇。BSON 是 MongoDB 内部用于序列化 JSON 文档的一种二进制数据格式。更多相关细节参考如下：

- BSON 标准；
- MongoDB 官方网站。

可以使用 BSON 向 JSON 文档中添加更加丰富的数据类型。

但考虑到本章的目的：

- 你只需要了解 JSON，即可访问数据库；
- JSON 是 MongoDB 向外暴露的接口，BSON 则只在 MongoDB 内部使用。

9.2 安装MongoDB

在继续深入前，我们先安装 MongoDB。相关操作可参考 A.4 节中的内容。准备好 MongoDB 后即可构建并运行本章中的示例。

9.3 MongoDB服务器及相关工具

MongoDB 由以下部件组成。

- MongoDB 服务器，即 `mongod`。
- 用 JavaScript 编写的命令行。
- 数据库驱动，该部件使得开发人员能够在自己的编程平台上访问 MongoDB。作为 MongoDB 的创始公司，10gen 支持很多编程语言，具体包括：Java、Ruby、JavaScript、Node.js、C++ 和 C#/.Net 等。关于官方支持的驱动列表，可访问 MongoDB 网站。
- 命令行工具。
  - `mongodump` 和 `mongorestore` 工具提供备份和恢复功能。
  - `mongoexport` 和 `mongoimport` 工具提供 MongoDB 的数据导入 / 导出功能，支持的数据类型包括 CSV、TSV 和 JSON。
  - `mongostat` 用于监听数据库性能（如连接数、内存使用情况等）。

9.4 MongoDB服务器

`mongod` 进程与其他数据库服务器类似，能接受连接并处理对数据的增删查改操作。可以在 macOS 和 Linux 的命令行中启动 `mongod`：

```
mongod &
```

如果 MongoDB 安装正确，则初始启动中的日志会显示如下：

```
2016-06-29T11:05:37.960-0600 I CONTROL  [initandlisten] MongoDB starting : pid...
2016-06-29T11:05:37.961-0600 I CONTROL  [initandlisten] db version v3.2.4
2016-06-29T11:05:37.961-0600 I CONTROL  [initandlisten] git version: e2ee9ffcf...
2016-06-29T11:05:37.961-0600 I CONTROL  [initandlisten] allocator: system
2016-06-29T11:05:37.961-0600 I CONTROL  [initandlisten] modules: none
2016-06-29T11:05:37.961-0600 I CONTROL  [initandlisten] build environment:
2016-06-29T11:05:37.961-0600 I CONTROL  [initandlisten]     distarch: x86_64
2016-06-29T11:05:37.961-0600 I CONTROL  [initandlisten]     target_arch: x86_64
```

```
2016-06-29T11:05:37.961-0600 I CONTROL  [initandlisten] options: { config: "/u...
2016-06-29T11:05:37.962-0600 I -        [initandlisten] Detected data files in...
2016-06-29T11:05:37.963-0600 W -        [initandlisten] Detected unclean shutd...
2016-06-29T11:05:37.973-0600 I JOURNAL  [initandlisten] journal dir=/usr/local...
2016-06-29T11:05:37.973-0600 I JOURNAL  [initandlisten] recover begin
2016-06-29T11:05:37.973-0600 I JOURNAL  [initandlisten] info no lsn file in jo...
2016-06-29T11:05:37.973-0600 I JOURNAL  [initandlisten] recover lsn: 0
2016-06-29T11:05:37.973-0600 I JOURNAL  [initandlisten] recover /usr/local/var...
2016-06-29T11:05:37.974-0600 I JOURNAL  [initandlisten] recover applying initi...
2016-06-29T11:05:37.976-0600 I JOURNAL  [initandlisten] recover cleaning up
2016-06-29T11:05:37.976-0600 I JOURNAL  [initandlisten] removeJournalFiles
2016-06-29T11:05:37.977-0600 I JOURNAL  [initandlisten] recover done
2016-06-29T11:05:37.996-0600 I JOURNAL  [durability] Durability thread started
2016-06-29T11:05:37.996-0600 I JOURNAL  [journal writer] Journal writer thread...
2016-06-29T11:05:38.329-0600 I NETWORK  [HostnameCanonicalizationWorker] Start...
2016-06-29T11:05:38.330-0600 I FTDC     [initandlisten] Initializing full-time...
2016-06-29T11:05:38.330-0600 I NETWORK  [initandlisten] waiting for connection...
2016-06-29T11:05:39.023-0600 I FTDC     [ftdc] Unclean full-time diagnostic da...
```

从外部看，MongoDB 会监听 27017 端口，但你可以通过以下选项更改该端口号：

```
mongod --port <your-port-number>
```

可以在命令行中输入以下命令来关闭服务器：

```
kill <pid>
```

该命令中的 `<pid>` 表示 `mongod` 进程中的进程 ID 号（Process ID，PID）。切记，不要使用 `kill -9` 来执行关闭操作，因为这么做会损坏数据库。

9.5　将JSON导入MongoDB

我们已经成功运行了 MongoDB 服务器，接下来会将演讲者数据导入数据库。使用 `mongoimport` 工具将 speakers.json 文件上传到 MongoDB。虽然我们一直在使用这些演讲者数据，但现在需要剥离数据的最外层文档结构以及 `speakers` 数组名：

```
{
  "speakers": [
  ]
}
```

调整后的 speakers.json 文件如例 9-1 所示。

例 9-1　speakers.json

```
[
  {
    "fullName": "Larson Richard",
    "tags": [
      "JavaScript",
      "AngularJS",
      "Yeoman"
    ],
```

```
    "age": 39,
    "registered": true
  }, {
    "fullName": "Ester Clements",
    "tags": [
      "REST",
      "Ruby on Rails",
      "APIs"
    ],
    "age": 29,
    "registered": true
  }, {
    "fullName": "Christensen Fisher",
    "tags": [
      "Java",
      "Spring",
      "Maven",
      "REST"
    ],
    "age": 45,
    "registered": false
  }
]
```

因为我们并不希望将 JSON 文件的内容以整个文档的形式插入数据库，所以这一调整是完全有必要的。如果不进行这样的调整，则导入结果将会是数据库中插入了单个的 speakers 数组文档。相反，我们需要的是 speaker 文档所组成的集合，且集合中的每个 speaker 文档与输入文件中的 speaker 对象一一对应。

在命令行中执行 mongoimport 后可以观察到以下结果：

```
json-at-work => mongoimport --db=jsaw --collection=speakers --upsert --jsonArray --file=speakers.json
2016-06-30T10:33:50.202-0600    connected to: localhost
2016-06-30T10:33:50.207-0600    imported 3 documents
json-at-work => mongo
MongoDB shell version: 3.2.4
connecting to: test
> use jsaw
switched to db jsaw
> db.speakers.find()
{ "_id" : ObjectId("577549ee061561f7f9be9725"), "fullName" : "Larson Richard", "tags" : [ "JavaScript", "AngularJS", "Ye
oman" ], "age" : 39, "registered" : true }
{ "_id" : ObjectId("577549ee061561f7f9be9726"), "fullName" : "Ester Clements", "tags" : [ "REST", "Ruby on Rails", "APIs
" ], "age" : 29, "registered" : true }
{ "_id" : ObjectId("577549ee061561f7f9be9727"), "fullName" : "Christensen Fisher", "tags" : [ "Java", "Spring", "Maven",
 "REST" ], "age" : 45, "registered" : false }
> 
```

在以上示例中，我们用到了以下工具。

- mongoimport 将 speakers JSON 文件中的数据导入 jsaw 数据库的 speakers 集合。
- mongo 访问 MongoDB，并选择 speakers 集合中的所有文档。更多细节信息可参考下节内容。

表 9-1 展示了 MongoDB 的基本概念与关系型数据库概念之间的映射关系。

表9-1：MongoDB与关系型数据库

MongoDB	关系型数据库
数据库	数据库实例
集合	表
文档	行

9.6 MongoDB命令行

在 MongoDB 中成功写入数据并运行后，现在是时候访问数据库并使用演讲者数据了。之前示例中所展示的 `mongo` 这一命令行工具提供了访问 MongoDB 的功能。可以使用以下方式来启动 `mongo`：

```
json-at-work => mongo
MongoDB shell version: 3.2.4
connecting to: test
>
```

`mongo` 默认连接的是 `test` 数据库，我们将使用另一个名为 `jsaw`（英文版书名 *JSON at Work* 的缩写）的数据库来单独存放演讲者数据：

```
json-at-work => mongo
MongoDB shell version: 3.2.4
connecting to: test
> use jsaw
switched to db jsaw
>
```

`use` 命令将当前数据库切换为 `jsaw`，切换后执行的所有命令都只会影响 `jsaw` 这一个数据库。你可能会好奇 `jsaw` 数据库是如何创建的，具体的创建方式有两种。

- 通过 `mongoimport` 工具创建。当执行导入操作时，`--db=jsaw` 和 `--collection=speakers` 命令行选项会创建 `jsaw` 数据库及其中的 `speakers` 集合。
- 通过 `mongo` 向集合中插入文档也可以创建 `jsaw` 数据库。后文将介绍相关操作。

可在提示符中输入 `exit` 来退出命令行。该操作会结束 MongoDB 的命令行交互，并返回操作系统控制台。

用mongo进行基本的增删改查操作

了解了 `mongo` 命令的一些基本操作后，接下来我们将使用 `mongo` 对演讲者数据进行增删改查操作。在命令行中使用的 MongoDB 查询语言是基于 JavaScript 的，这有效简化了访问 JSON 文档的操作。

1. 查询文档

以下是获取 `speakers` 集合（本章前面已经将其导入 MongoDB）中的所有文档的操作：

```
json-at-work => mongo jsaw
MongoDB shell version: 3.2.4
connecting to: jsaw
> db.speakers.find()
{ "_id" : ObjectId("577549ee061561f7f9be9725"), "fullName" : "Larson Richard", "tags" : [ "JavaScript", "AngularJS", "Yeoman" ]
, "age" : 39, "registered" : true }
{ "_id" : ObjectId("577549ee061561f7f9be9726"), "fullName" : "Ester Clements", "tags" : [ "REST", "Ruby on Rails", "APIs" ], "a
ge" : 29, "registered" : true }
{ "_id" : ObjectId("577549ee061561f7f9be9727"), "fullName" : "Christensen Fisher", "tags" : [ "Java", "Spring", "Maven", "REST"
 ], "age" : 45, "registered" : false }
>
```

有关该命令（`db.speakers.find()`）的详细说明如下。

- 操作命令以 `db` 开头。
- `speakers` 是集合的名称。
- 在没有任何查询参数的情况下，`find()` 会返回 `speakers` 集合中的所有文档。

再看一下命令行执行的结果，我们可以看到返回的数据与 JSON 类似，几乎可以说是一模一样。复制执行结果，并粘贴到 JSONLint 中。点击 Validate JSON 按钮，你会发现 `_id` 字段有误。当 `mongoimport` 命令导入 JSON 输入文件中的演讲者数据并创建 `speakers` 集合时，MongoDB 会在每个文档中插入 `_id` 字段（用作数据库中的主键的对象 ID）。出于以下原因，MongoDB 命令行中的查询结果并不是合法的 JSON。

- 最外面表示数组的中括号（`[]`）缺失。
- `ObjectId(...)` 并不是合法的 JSON 值。JSON 中的合法标量值包括：数值、布尔值，以及用双引号括起来的字符串。
- 用于分隔 `speaker` 文档的逗号缺失。

本章前面介绍了如何将合法的 JSON 导入 MongoDB，介绍完剩余的增删改查操作后，我们会展示如何将 MongoDB 集合导出为合法的 JSON。

如果只需要返回标签中包含 REST 的演讲者，可以在 `find()` 方法中添加查询参数：

```
json-at-work => mongo jsaw
MongoDB shell version: 3.2.4
connecting to: jsaw
> db.speakers.find({tags:'REST'})
{ "_id" : ObjectId("577549ee061561f7f9be9726"), "fullName" : "Ester Clements", "tags" : [ "REST", "Ruby on Rails", "APIs" ], "a
ge" : 29, "registered" : true }
{ "_id" : ObjectId("577549ee061561f7f9be9727"), "fullName" : "Christensen Fisher", "tags" : [ "Java", "Spring", "Maven", "REST"
 ], "age" : 45, "registered" : false }
>
```

我们在这个示例中添加了查询参数 `{tags:'REST'}`，该参数使得查询结果只返回 `tags` 数组中包含 `'REST'` 值的那些 `speaker` 文档。MongoDB 查询语言是基于 JavaScript 对象字面量语法设计的。如需提高有关 JavaScript 对象方面的知识，可参考 David Flanagan 所著的《JavaScript 权威指南（第 6 版）》。

可以使用以下命令来获取 `speakers` 集合中的文档数量：

```
> db.speakers.count()
3
```

2. 创建文档

以下示例展示了如何在 `speakers` 集合中添加新的文档：

```
json-at-work => mongo jsaw
MongoDB shell version: 3.2.4
connecting to: jsaw
> db.speakers.insert({
...     fullName: 'Carl ClojureDev',
...     tags: ['Clojure', 'Functional Programming'],
...     age: 45,
...     registered: false
... })
WriteResult({ "nInserted" : 1 })
> db.speakers.find()
{ "_id" : ObjectId("577549ee061561f7f9be9725"), "fullName" : "Larson Richard", "tags" : [ "JavaScript", "AngularJS", "Yeoman" ]
, "age" : 39, "registered" : true }
{ "_id" : ObjectId("577549ee061561f7f9be9726"), "fullName" : "Ester Clements", "tags" : [ "REST", "Ruby on Rails", "APIs" ], "a
ge" : 29, "registered" : true }
{ "_id" : ObjectId("577549ee061561f7f9be9727"), "fullName" : "Christensen Fisher", "tags" : [ "Java", "Spring", "Maven", "REST"
 ], "age" : 45, "registered" : false }
{ "_id" : ObjectId("577584327a0be85396f1daed"), "fullName" : "Carl ClojureDev", "tags" : [ "Clojure", "Functional Programming"
], "age" : 45, "registered" : false }
>
```

该示例使用了 `insert()` 函数，在调用函数时传入了包含名称 – 值对的 JavaScript 对象字面量，从而创建了新的 `speaker` 文档。

3. 修改文档

我们的新演讲者 Carl ClojureDev 决定在自己的技术擅长点上增加 Scala。为了将该编程语言添加到 `tags` 数组中，需要执行以下操作：

```
json-at-work => mongo jsaw
MongoDB shell version: 3.2.4
connecting to: jsaw
> db.speakers.find({fullName: 'Carl ClojureDev'})
{ "_id" : ObjectId("577584327a0be85396f1daed"), "fullName" : "Carl ClojureDev", "tags" : [ "Clojure" ], "age" : 45, "registered
" : false }
> db.speakers.update({fullName: 'Carl ClojureDev'},
... { $push:
...     { tags: 'Scala' }
... })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.speakers.find({fullName: 'Carl ClojureDev'})
{ "_id" : ObjectId("577584327a0be85396f1daed"), "fullName" : "Carl ClojureDev", "tags" : [ "Clojure", "Scala" ], "age" : 45, "r
egistered" : false }
>
```

该示例使用了 `update()` 函数，具体解释如下。

- `{fullName: 'Carl ClojureDev'}` 查询参数负责找到需要进行修改的那个 `speaker` 文档。
- `$push` 操作符在 `tags` 数组中添加 `'Scala'`。这与 JavaScript 中的 `push()` 函数比较像。

值得注意的是，`update()` 函数中还能使用很多其他的操作符，如 `$set`；但 `$set` 会用一个全新的值来覆盖目标字段，因此使用时需要格外小心。

4. 删除文档

最后，从集合中删除 Carl ClojureDev 这一演讲者：

```
json-at-work => mongo jsaw
MongoDB shell version: 3.2.4
connecting to: jsaw
> db.speakers.find({fullName: 'Carl ClojureDev'})
{ "_id" : ObjectId("5775906647776536ff96a2fc"), "fullName" : "Carl ClojureDev", "tags" : [ "Clojure", "Scala", "Functional Prog
ramming" ], "age" : 45, "registered" : false }
> db.speakers.remove({fullName: 'Carl ClojureDev'})
WriteResult({ "nRemoved" : 1 })
> db.speakers.find({fullName: 'Carl ClojureDev'})
> db.speakers.find()
{ "_id" : ObjectId("577549ee061561f7f9be9725"), "fullName" : "Larson Richard", "tags" : [ "JavaScript", "AngularJS", "Yeoman" ]
, "age" : 39, "registered" : true }
{ "_id" : ObjectId("577549ee061561f7f9be9726"), "fullName" : "Ester Clements", "tags" : [ "REST", "Ruby on Rails", "APIs" ], "a
ge" : 29, "registered" : true }
{ "_id" : ObjectId("577549ee061561f7f9be9727"), "fullName" : "Christensen Fisher", "tags" : [ "Java", "Spring", "Maven", "REST"
 ], "age" : 45, "registered" : false }
>
```

在该示例中，我们使用含有 {fullName: 'Carl ClojureDev'} 参数的 remove() 函数来删除特定文档。如果之后调用 find() 函数，就可以发现 speakers 集合中的该文档确实已经删除，其他文档则不受影响。

9.7 从MongoDB中导出JSON文档

了解了 MongoDB 服务器和命令行的使用后，接下来我们介绍如何将数据导出为合法的 JSON 文档。按照以下方式使用 mongoexport 工具即可看到相应结果：

```
json-at-work => mongoexport --db=jsaw --collection=speakers --pretty --jsonArray
2016-06-30T12:58:32.270-0600    connected to: localhost
[{
        "_id": {
                "$oid": "577549ee061561f7f9be9725"
        },
        "fullName": "Larson Richard",
        "tags": [
                "JavaScript",
                "AngularJS",
                "Yeoman"
        ],
        "age": 39,
        "registered": true
},
{
        "_id": {
                "$oid": "577549ee061561f7f9be9726"
        },
        "fullName": "Ester Clements",
        "tags": [
                "REST",
                "Ruby on Rails",
                "APIs"
        ],
        "age": 29,
        "registered": true
},
{
        "_id": {
                "$oid": "577549ee061561f7f9be9727"
        },
        "fullName": "Christensen Fisher",
        "tags": [
                "Java",
                "Spring",
                "Maven",
                "REST"
        ],
        "age": 45,
        "registered": false
}]

2016-06-30T12:58:32.271-0600    exported 3 records
```

以上示例中的 mongoexport 命令读取 jsaw 数据库内的 speakers 集合中的数据，并将 JSON 数组优化显示在标准输出中。这是一个不错的开始，但为了获取能在 MongoDB 外正常使用的合法 JSON，我们还需要移除结果中的 MongoDB 对象 ID（_id）。mongoexport 始终会返回 _id，因此我们需要借助其他工具来过滤 _id 字段。

通过组合使用工具，我们可以获取最终需要的 JSON 格式，而 jq 是满足这一点的完美选择。你或许还记得第 6 章中提到过的内容，jq 是一个非常优秀的命令行工具，除了搜索 JSON，它还提供了高质量的 JSON 过滤功能。jq 不像第 7 章中提到的 Handlebars 那样具备完整的 JSON 转换功能，但也足够满足我们的需求了。通过管道将 mongoexport 命令的结果输入 jq 以进行处理，我们可以观察到以下结果：

```
json-at-work => mongoexport --db=jsaw --collection=speakers --pretty --jsonArray | jq '[.[] | del(._id)]'
2016-06-30T13:09:56.236-0600    connected to: localhost
2016-06-30T13:09:56.237-0600    exported 3 records
[
  {
    "fullName": "Larson Richard",
    "tags": [
      "JavaScript",
      "AngularJS",
      "Yeoman"
    ],
    "age": 39,
    "registered": true
  },
  {
    "fullName": "Ester Clements",
    "tags": [
      "REST",
      "Ruby on Rails",
      "APIs"
    ],
    "age": 29,
    "registered": true
  },
  {
    "fullName": "Christensen Fisher",
    "tags": [
      "Java",
      "Spring",
      "Maven",
      "REST"
    ],
    "age": 45,
    "registered": false
  }
]
json-at-work =>
```

最终结果完全符合我们的预期，即由不包含 MongoDB 对象 ID 的 speaker 对象所组成的合法 JSON 数组。以下是对该命令的详细解释。

- 有关 mongoexport 命令的解释如下。
  - --db=jsaw --collection=speakers 声明了读取的是 jsaw 数据库中的 speakers 集合。
  - --pretty --jsonArray 确保输出的是一个优化显示的 JSON 数组。
- mongoexport 的结果被导入标准输出，随后通过管道输入 jq。
- jq 表达式 [.[] | del(._id)] 的工作机制如下。

– 最外层的数组中括号（[]）确保输入的 JSON 数组、对象及其字段在最终输出结果中保持不变。
– .[] 声明 jq 会搜索整个数组。
– 目标为 del(._id) 命令的管道声明 jq 将从输出中删除所有的 _id 字段。

- jq 的结果被导入标准输出，这可以作为输入写入文件。

该实际示例展示了 jq 强大的功能。虽然语法有些过于精炼，但 jq 还是 JSON 工具集中一个很不错的选择。如需了解更多有关 jq 的信息，可参考第 6 章中的相关内容。另外，也可查阅 jq 手册。

9.8 关于Schema

MongoDB 是没有 Schema 的，这意味着数据库既不会校验数据，也不会在存储数据时依赖 Schema。然而，应用程序对存储在每个文档中的数据还是会有数据结构上的预期，因为只有这样，应用程序才能放心地使用集合与文档。对象文档映射（Object Document Mapper，ODM）在 MongoDB 之上提供了额外的特性：

- 用于校验数据并强制使用统一数据结构的 Schema；
- 对象建模；
- 基于对象的数据访问。

MongoDB 中不存在单个的跨平台 ODM。相反，每个编程平台都有自己的相关类库。Node.js 开发者一般会使用 Mongoose。以下是一个简要的示例，展示了如何声明 speaker 的 Schema、如何创建模型，以及如何向数据库中插入 speaker 对象。

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
mongoose.connect('mongodb://localhost/jsaw');

// 声明演讲者Schema

var speakerSchema = new Schema({
  fullName:  String,
  tags: [String],
  age: Number,
  registered: Boolean
});

// 创建演讲者模型

var Speaker = mongoose.model('Speaker', speakerSchema);

var speaker = new Speaker({
  fullName: 'Carl ClojureDev',
  tags: ['Clojure', 'Functional Programming'],
  age: 45,
  registered: false
```

```
});
speaker.save(function (err) {
  if (err) {
    console.log(err);
  } else {
    console.log('Created Speaker: ' + speaker.fullName);
  }
});
```

Mongoose 模型的实质是一个基于 Schema 的构造器，该构造器封装了访问 MongoDB 集合的底层细节。Mongoose 文档则是 Mongoose 模型的一个实例，提供了访问 MongoDB 文档的功能。Mongoose Schema 和 JSON Schema 是两个不同的东西。Node.js 模块 `json-schema-to-mongoose` 可以将 JSON Schema 转换为对应的 Mongoose Schema，具体操作就留给你自己实践了。除了创建文档，Mongoose 还提供了操作文档的其他功能：读取（`find()`）、修改（`save()` 或 `update()`）和删除（`remove()`）。

其他编程平台也有自己的访问 MongoDB 的 ODM。

Java
: Spring 用户可以使用提供了 POJO 和 MongoDB 映射的 Spring Data，Hibernate OGM 则为包括 MongoDB 在内的 NoSQL 数据库提供了 Java 持久化 API 支持。

Ruby
: 可以使用 MongoDB 官方支持的 Mongoid。

9.9 用MongoDB进行RESTful API测试

完整地介绍 MEAN 技术栈超出了本书的探讨范围，这么做的话就偏离了本章的主题，并且无法专注于 JSON。接下来我们会对 MongoDB 进行不同的操作，将其作为模拟的 RESTful API 来使用。使用模拟 RESTful API 的优势是显著的。

- 无须编码，这将开发者从开发、维护基础设施代码这样单调沉闷的工作中解放出来了。这样一来，开发人员就能够专注于提供业务价值（API 中的业务逻辑）的那部分代码。
- 促使 API 开发团队在正式开始编码**前**先创建一个 API 的初始设计。这一策略称为“API 优先”设计。这么做的话，因为模拟 API 中没有任何实现，开发者只是基于这一接口进行后续的设计，所以最终的 API 就不太会向外暴露领域对象与数据库中的具体实现细节。
- API 的使用者无须等待真正的 API 完成即可拥有一个可行的模拟替代版本。
- API 的开发者现在拥有足够的时间来进行开发，无须为了支持 API 的使用者而赶工发布。
- API 的开发者可以尽早从使用者那儿收集有关 API 可用性的反馈，并使用这些信息来迭代更新设计与实现。

9.9.1 测试输入数据

我们将继续使用本章前面所导入的演讲者数据。

9.9.2 对MongoDB进行RESTful封装

根据 MongoDB 的文档，有好几个可靠的 REST 接口工具以独立服务器的形式运行在 MongoDB 之上，具体如下。

Crest
: 基于 Node.js 的 Crest 能够提供完整的增删改查操作（HTTP `GET`、`PUT`、`POST` 和 `DELETE`），可在其 GitHub 主页上找到更多细节信息。

RESTHeart
: 基于 Java 的 RESTHeart 能够提供完整的增删改查功能，可查看其主页了解更多信息。

DrowsyDromedary
: 基于 Ruby 的 DrowsyDromedary 能够提供完整的增删改查功能，可在其 GitHub 主页上查看更多信息。

Simple REST API
: MongoDB 默认提供该工具，但该工具只支持 HTTP `GET`，不提供完整的 REST 功能（`PUT`、`POST` 和 `DELETE`）。如需了解更多信息，可参考 RESTHeart 网站上的 Simple REST API 文档。

因为 Crest、RESTHeart 和 DrowsyDromedary 都支持主要的几个 HTTP 方法，能够处理使用者的增删改查请求，所以它们都可以满足我们的需求。考虑到 Crest 易于安装配置，因此接下来我们将主要使用 Crest。可参考 A.2.5 节中的内容来安装 Crest。然后在本机中切换到 crest 目录，在命令行中使用 `node server` 来启动 Crest 服务器。你应该可以观察到以下结果：

```
node server

DEBUG: util.js is loaded
DEBUG: rest.js is loaded
crest listening at http://:::3500
```

然后打开浏览器并访问以下 URL：http://localhost:3500/jsaw/speakers。该访问请求会使得 Crest 对 MongoDB 中 `jsaw` 数据库内的 `speakers` 集合执行 `GET`（读取 / 搜索）操作。访问后可看到如图 9-1 所示的结果。



图 9-1：由 MongoDB/Crest 提供的可在浏览器中访问的演讲者数据

这是一个不错的开始，但因为浏览器只能发送 HTTP `GET` 请求，所以无法在浏览器中进行完整的 API 测试。接下来，我们将使用之前章节中提到的 Postman 来完整测试 Crest/MongoDB 提供的演讲者数据 API。输入 URL：http://localhost:3500/jsaw/speakers，选择 HTTP 方法为 `GET`，最后点击 Send 按钮。应该可以看到如图 9-2 所示的结果。

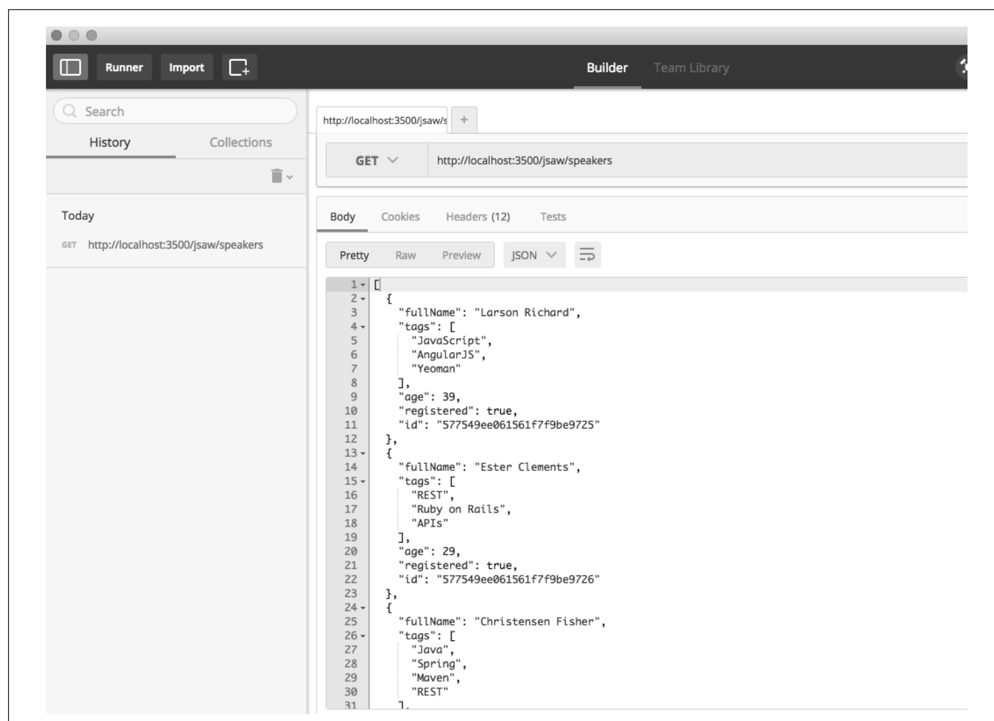


图 9-2：由 MongoDB/Crest 提供的可在 Postman 中访问的演讲者数据

这与之前在浏览器中看到的结果相同，但这一次我们可以修改 API 所提供的数据了。接下来，我们删除其中一个 `speaker` 对象。首先，复制其中一个 `speaker` 对象的 `id`，将其添加到 URL 中：http://localhost:3500/jsaw/speakers/id（URL 中的 `id` 即为刚复制的对象 ID）。然后在 Postman 中选择 `DELETE` 作为 HTTP 方法，并点击 Send 按钮，随后即可观察到以下 HTTP 响应：

```
{
  "ok": 1
}
```

接着重复 `DELETE` 前的操作，再一次用 `GET` 访问 http://localhost:3500/jsaw/speakers，可以确认 Crest 的确调用了 MongoDB 来删除选中的 `speaker`。

至此，在不编写任何代码和搭建任何基础设施的情况下，我们拥有了一个具备完整功能的模拟 REST API，该 API 可以访问 MongoDB 数据库并返回合法的 JSON 输出。使用这种风格的工作流来梳理 API 的设计与测试吧，你将发现团队的生产力会得到巨大提高。

9.10　本章回顾

本章介绍了 JSON 与 MongoDB 协作的一些基本知识，具体包括：

- 将 JSON 文档导入 MongoDB；
- 在 MongoDB 中执行主要的几个增删改查操作；
- 将 MongoDB 中的数据导出为 JSON 文档；
- 在不编写代码的情况下，以模拟 RESTful API 的形式访问 MongoDB。

9.11 内容预告

介绍完 JSON 与 MongoDB 之间的协作后，我们将转而讨论 JSON 企业级应用中的最后一个话题，探讨 JSON 在 Apache Kafka 中的使用情况。

第10章

用Kafka实现JSON消息系统

Apache Kafka 是一个流行的分布式可扩展的消息系统，允许（在不同编程平台上运行的）异构应用程序以消息传递的方式进行异步通信。Kafka 一开始是 LinkedIn 工程团队开发出来的，属于某个主要产品架构调整的一部分。从大型单体应用程序转型为微服务架构模式后，LinkedIn 公司创建了 Kafka，以提供通用的可处理大型消息体的数据流功能，从而在整个公司范围内整合服务和应用程序。2011 年，LinkedIn 向 Apache 基金会开源了 Kafka。如今，很多公司都成功使用 Kafka 作为企业产品架构中的消息平台中心。如需了解更多有关 Kafka 的信息，可参考 Apache 的 Kafka 主页。

Kafka 与其他消息系统（如 Java Message Service，JMS）的不同之处在于：Kafka 与具体的编程平台无关。虽然 Kafka 本身是用 Java 编写的，但消息的生产者和消费者可以使用其他编程语言来开发。为了演示这一点，在本章的端到端示例中，我们会使用 Node.js 和 Bourne Shell 来编写消息的消费者。

Kafka 既支持二进制消息，也支持文本消息。最受欢迎的文本消息格式有：普通文本、JSON 和 Apache Avro。Kafka 的 API（消息的生产者和消费者所用的接口）是基于 TCP 进行通信的。本章将 Kafka 作为传统的 JSON 消息系统，并展示以下操作：

- 在命令行中通过 Kafka 来生产 / 使用 JSON 消息；
- 设计并实现一个小型的端到端示例，并在示例中使用 Kafka 与 JSON。

10.1 Kafka的用例

典型的 Kafka 用例如下所示。

传统的消息系统

应用程序发布其他应用程序所使用的消息。Kafka 使用异步（发送者无须等待响应结果）的发布 / 订阅模型来解耦消息的生产者和消费者。

分析与流处理

应用程序向 Kafka 的相关主题发布实时的使用信息（如鼠标点击、访客信息、会话、页面访问和购买操作）。然后，像 Apache Spark/Spark Streaming 这样的流处理应用程序就可以读取多个主题的消息、转换数据（通过 map/reduce 等），并通过 Flume 将最终的结果发送到 Hadoop 这样的数据仓库中。对于目标数据仓库中的数据，你可以使用各种分析工具（如数据可视化）来处理。

运维与应用程序性能监控

应用程序可以发布各种统计数据（如消息数、事务数、响应时间、HTTP 响应码和 HTTP 请求数等），运维人员则可以对此进行检查、监控，从而追踪性能、产品使用和潜在问题。

日志归集

企业中的所有应用程序都可以向某个 Kafka 主题发布日志消息，然后就可以使用日志管理应用程序对它们进行处理，如 ELK（ElasticSearch、Logstash、Kibana）技术栈。Kafka 可以部署在 Logstash 前来接收海量数据，并允许 Logstash 在不损失信息的情况下以自己的节奏来执行那些性能消耗较大的操作。

10.2　Kafka中的概念和专有名词

以下是 Kafka 架构中的一些关键概念。

生产者

负责向主题发布消息。

消费者

注册、订阅主题，读取出现的消息。

主题

一个命名的频道，某一类消息的订阅源。在本章的示例中，`new-proposals-recvd` 主题包含了 MyConference 中的新演说提案的所有消息。还可以将主题看作业务事件所组成的流，流中包括了下订单和退货等事件消息。一个主题可以分为一个或多个分区。

代理

管理一个或多个主题的 Kafka 服务器。

集群

包含一个或多个代理。

分区

在分布式环境中，一个主题在多个分区中拥有副本，每个副本都由单独的代理所管理。

偏移量

分区中的消息所独有的 ID。Kafka 使用偏移量来维护消息的顺序。

了解这些概念后，就可以在本章中进行 JSON 消息的生产和消费了。为简洁起见，本书不涉及 Kafka 中其他的一些重要领域，其中包括：耐久性、消费者组、消息传递保障和副本。Kafka 是一个可以用整本书来讲述的大话题，如需了解更多信息，可参考 Neha Narkhede 所著的《Kafka 权威指南》[1]。

在本章的示例中，我们会使用单个代理（Kafka 服务器），而每个主题也都只有一个分区。

10.3 Kafka生态系统——相关项目

Kafka 是一个通用的消息系统，可与其他消息处理系统集成来搭建更为强大的消息应用程序。Kafka 的技术生态系统包括但不局限于以下项目。

Apache Spark/Spark Streaming

用于流处理（详见 10.1 节）。

HiveKa

与 Hive 进行集成，从而为 Kafka 的主题提供类 SQL 的接口。

ElasticSearch

独立的消费者从 Kafka 的主题中拉取数据，然后加载到 ElasticSearch 中进行处理。

Kafka Manager

Kafka 的管理控制台，可用于管理 Kafka 集群、主题、消费者等。

Flume

将大量数据从频道（如 Kafka 主题）迁移到 Hadoop 分布式文件系统中。

Avro

一种数据序列化格式，可以作为纯 JSON 格式的替代来提供更丰富的数据结构。Avro 不是一项标准，但拥有自己的 JSON 格式的 Schema（该 Schema 与 JSON Schema 无关）。作为 JSON 的替代方案，Avro 能提供更丰富的数据结构和更紧凑的数据格式。Avro 一开始是 Hadoop 的一部分，但最终成为了一个单独的项目。

以上列举的项目仅仅只是能与 Kafka 协作的小部分系统。有关完整的 Kafka 生态系统的描述，可参考 Kafka 生态系统网页。

10.4 配置Kafka环境

在介绍 Kafka 的命令行界面前，需要先安装 Kafka 和 Apache ZooKeeper 来构建并运行本章中的所有示例。可参考 A.8 节中的内容来安装 Kafka 和 ZooKeeper。

注 1：此书已由人民邮电出版社出版。——编者注

安装好 Kafka 后，就可以对其进行配置，以允许删除主题（该操作默认是禁用的）。按以下方式编辑 KAFKA-INSTALL-DIR/KAFKA_VERSION/libexec/config/server.properties 文件（KAFKA-INSTALL-DIR 是 Kafka 的安装目录，KAFKA_VERSION 则是安装的 Kafka 版本）：

```
# 允许/禁止主题删除的开关，默认值为false
delete.topic.enable=true
```

为什么需要ZooKeeper

至此，你可能会产生一些困惑：为什么除 Kafka 外还需要 ZooKeeper 呢？简单来说就是 Kafka 需要 ZooKeeper 才能运行。换而言之，作为分布式应用程序的 Kafka 就是设计在 ZooKeeper 环境中运行的。ZooKeeper 是一个用于协调分布式进程的服务器，具体根据以下参数来进行管理：命名、状态信息、配置、位置信息、同步状态、容错转移等；其命名注册表使用了与文件系统相似的层级化命名空间策略。

很多知名项目都使用了 ZooKeeper，其中包括 Kafka、Storm、Hadoop、MapReduce、HBase 和 Solr（云服务版）。如需学习更多相关知识，可访问 ZooKeeper 的官方网站。

10.5 Kafka命令行界面

Kafka 具有内置的命令行界面，这允许开发人员对 Kafka 进行一些实验操作。我们将展示如何启动 Kafka、如何发布 JSON 消息，以及如何关闭 Kafka 程序。

如需使用已经编写好的脚本以避免大量的手动输入，可以访问本书代码示例中的 chapter-10/scripts 目录，然后更改文件权限，将所有的脚本设置为可执行：

```
chmod +x *.sh
```

10.5.1 如何用命令行界面发布JSON消息

以下是启动 Kafka 并发布 / 使用消息的一系列步骤（按操作顺序排列）。

(1) 启动 ZooKeeper。
(2) 启动 Kafka 服务器。
(3) 创建主题。
(4) 启动消费者程序。
(5) 向主题发布消息。
(6) 使用该消息。
(7) 清理并关闭 Kafka。
- 关闭消费者程序。
- 删除主题。
- 关闭 Kafka。
- 关闭 ZooKeeper。

10.5.2 启动ZooKeeper

正如之前所提到的，Kafka 依赖 ZooKeeper 才能运行。可以在一个新的命令行终端中运行以下命令来启动 ZooKeeper：

```
./start-zookeeper.sh
```

例 10-1 展示了该脚本的内容。

例 10-1 scripts/start-zookeeper.sh

```
zkServer start
```

运行后可以观察到以下结果：

```
json-at-work => ./start-zookeeper.sh
ZooKeeper JMX enabled by default
Using config: /usr/local/etc/zookeeper/zoo.cfg
Starting zookeeper ... STARTED
```

10.5.3 启动Kafka

接下来，我们就可以在一个新的命令行终端中启动 Kafka 服务器了：

```
./start-kafka.sh
```

该脚本的内容如例 10-2 所示。

例 10-2 scripts/start-kafka.sh

```
kafka-server-start /usr/local/etc/kafka/server.properties
```

在该脚本中，server.properties 文件保存了有关 Kafka 的配置信息。为了允许删除主题，我们之前已经编辑过这一文件。

现在 Kafka 服务器应该已经成功运行了。执行这一命令会打印很多日志信息，启动完成后则可以看到以下结果：

```
[2016-12-31 16:42:01,371] INFO Creating /brokers/ids/0 (is it secure? false) (kafka.utils.ZKCheckedEphemeral)
[2016-12-31 16:42:01,375] INFO Result of znode creation is: OK (kafka.utils.ZKCheckedEphemeral)
[2016-12-31 16:42:01,377] INFO Registered broker 0 at path /brokers/ids/0 with addresses: PLAINTEXT -> EndPoint(10.229.1
04.161,9092,PLAINTEXT) (kafka.utils.ZkUtils)
[2016-12-31 16:42:01,385] INFO Kafka version : 0.10.1.0 (org.apache.kafka.common.utils.AppInfoParser)
[2016-12-31 16:42:01,385] INFO Kafka commitId : 3402a74efb23d1d4 (org.apache.kafka.common.utils.AppInfoParser)
[2016-12-31 16:42:01,386] INFO [Kafka Server 0], started (kafka.server.KafkaServer)
```

10.5.4 创建主题

接下来，我们创建主题 `test-proposals-recvd` 来接收新的演讲提案。可以通过在新的命令行终端中运行以下脚本来创建主题：

```
./create-topic.sh test-proposals-recvd
```

该脚本会运行 `kafka-topics` 命令，如例 10-3 所示。

例 10-3　scripts/create-topic.sh

```
...

kafka-topics --zookeeper localhost:2181 --create \
             --topic $1 --partitions 1 \
             --replication-factor 1
```

该脚本的工作机制如下。

- `$1` 是命令行变量，用于保存主题的名称（本例中为 `test-proposals-recvd`）。
- 为保持示例的简洁性，对于该主题，我们只使用一个分区（有序的记录列表）和一个副本。一个分区可以在多个服务器间进行复制，从而实现容错和负载均衡。在生产环境中，一般会有多个副本来支持大量的消息数据。

运行之前的脚本后可以观察到以下结果：

```
json-at-work => ./create-topic.sh test-proposals-recvd
Created topic "test-proposals-recvd".
```

10.5.5　列举主题

运行以下脚本可以确认已经成功创建新的主题：

```
./list-topics.sh
```

该脚本使用了 `kafka-topics` 命令，如例 10-4 所示。

例 10-4　scripts/list-topics.sh

```
kafka-topics --zookeeper localhost:2181 --list
```

可以看到，确实已经创建 `test-proposals-recvd` 主题：

```
json-at-work => ./list-topics.sh
__consumer_offsets
test-proposals-recvd
```

`__consumer_offsets` 是 Kafka 内部实现中的底层细节，因此无须关心。我们只需要关心刚创建的主题。

10.5.6　启动消费者程序

拥有主题后就可以生产和消费消息了。首先，我们使用以下脚本来创建一个监听 `test-proposals-recvd` 主题的消费者：

```
./start-consumer.sh test-proposals-recvd
```

该脚本使用了 `kafka-console-consumer` 命令，如例 10-5 所示。

例 10-5　scripts/start-consumer.sh

```
...

kafka-console-consumer --bootstrap-server localhost:9092 \
                       --topic $1
```

在该脚本中，作为命令行变量的 `$1` 保存了消费者监听的主题名称（本例中为 `test-proposals-recvd`）。

可以看到，消费者程序正在等待新消息，因此屏幕上暂时没有任何显示：

```
json-at-work => ./start-consumer.sh test-proposals-recvd
█
```

10.5.7　发布JSON消息

接下来就可以在新的命令行终端中使用以下脚本针对我们的主题来发布 JSON 消息了：

```
./publish-message.sh '{ "message": "This is a test proposal." }' test-proposals-
recvd
```

例 10-6 展示了该脚本的内容。

例 10-6　scripts/publish-message.sh

```
...

echo $MESSAGE_FROM_CLI | kafka-console-producer \
          --broker-list localhost:9092 \
          --topic $TOPIC_NAME_FROM_CLI

...
```

在以上脚本中，需要注意以下几点。

- 我们使用 `echo` 命令将 JSON 消息打印到标准输出中，并通过管道将其用作 `kafka-console-producer` 命令的输入。
- `$MESSAGE_FROM_CLI` 是命令行变量，保存了需要发布的 JSON 消息。
- `$TOPIC_NAME_FROM_CLI` 是命令行变量，保存了主题的名称（本例中为 `test-proposals-recvd`）。

发布消息后，应该可以观察到以下结果：

```
json-at-work => ./publish-message.sh '{ "message": "This is a test proposal." }' test-proposals-recvd
```

消息本身不会在用于发布的命令行终端中显示。

10.5.8　使用JSON消息

切换回启动消费者程序的命令行窗口，你应该可以看到消费者程序读取并打印了 `test-proposals-recvd` 主题中的消息：

```
json-at-work => ./start-consumer.sh test-proposals-recvd
{ "message": "This is a test proposal." }
█
```

至此，一个用于生产和消费 JSON 消息的简单 Kafka 示例就算完成了。接下来，我们将对这一示例进行清理扫尾工作。

10.5.9　清理并关闭Kafka

以下是清理和关闭 Kafka 的一系列步骤。

(1) 关闭消费者程序。
(2) 删除主题（可选）。
(3) 关闭 Kafka。
(4) 关闭 ZooKeeper。

1. 关闭消费者程序

只需在启动消费者程序的命令行窗口中按下 Ctrl-C，即可看到以下结果：

```
json-at-work => ./start-consumer.sh test-proposals-recvd
{ "message": "This is a test proposal." }
^CProcessed a total of 1 messages
```

2. 删除主题

可以使用以下脚本来删除 `test-proposals-recvd` 主题（此步骤为可选）：

```
./delete-topic.sh test-proposals-recvd
```

例 10-7 显示了该脚本的内容。

例 10-7　scripts/delete-topic.sh

```
...

kafka-topics --zookeeper localhost:2181 --delete --topic $1
```

在该脚本中，作为命令行变量的 `$1` 保存了主题的名称（本例中为 `test-proposals-recvd`）。

运行脚本后即可看到以下结果：

```
json-at-work => ./delete-topic.sh test-proposals-recvd
Topic test-proposals-recvd is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
```

3. 关闭Kafka

只需在启动 Kafka 的命令行窗口中按下 Ctrl-C，即可关闭 Kafka；你也可以使用以下命令来优雅地执行关闭操作：

```
./stop-kafka.sh
```

例 10-8 展示了该脚本的内容。

例 10-8　scripts/stop-kafka.sh

```
kafka-server-stop
```

该脚本使用了 `kafka-server-stop` 命令来关闭 Kafka 服务器。这一受控的关闭操作需要一定的时间，并会打印出很多日志消息。如果切换回启动 Kafka 服务器的命令行窗口，你应该可以在命令行的最后观察到以下消息：

```
[2016-12-31 18:40:06,981] INFO [GroupCoordinator 0]: Shutdown complete. (kafka.coordinator.GroupCoordinator)
[2016-12-31 18:40:06,988] INFO Terminate ZkClient event thread. (org.I0Itec.zkclient.ZkEventThread)
[2016-12-31 18:40:06,990] INFO Session: 0x159573c11390007 closed (org.apache.zookeeper.ZooKeeper)
[2016-12-31 18:40:06,990] INFO EventThread shut down for session: 0x159573c11390007 (org.apache.zookeeper.ClientCnxn)
[2016-12-31 18:40:06,992] INFO [Kafka Server 0], shut down completed (kafka.server.KafkaServer)
```

如果在之前的操作过程中删除了 `test-proposals-recvd` 主题，那么重新启动 Kafka 后该主题将不复存在。如果并未删除该主题，则重新启动 Kafka 后仍旧可以对该主题进行使用。

4. 关闭ZooKeeper

我们以关闭 ZooKeeper 来收尾。在命令行中输入以下命令：

```
./stop-zookeeper.sh
```

例 10-9 显示了该脚本的内容。

例 10-9　scripts/stop-zookeeper.sh

```
zkServer stop
```

至此，有关 Kafka 的所有程序都应该已经关闭了。你应该可以在命令行中看到以下结果：

```
json-at-work => ./stop-zookeeper.sh
ZooKeeper JMX enabled by default
Using config: /usr/local/etc/zookeeper/zoo.cfg
Stopping zookeeper ... STOPPED
```

10.6　Kafka的类库

Kafka 在主流的应用程序开发平台上有着广泛支持，具体包括以下类库。

Java
: Spring 在 Java 社区中广泛用于系统集成，并可以通过 Spring Kafka 为 Kafka 提供支持。

Ruby
: 可以在 GitHub 上找到 `karafka` gem。

JavaScript
: 在接下来的内容中，我们将使用 `kafka-node` 模块来搭建端到端示例。如需了解更多有关 `kafka-node` 的信息，可参考其 npm 和 GitHub 网站。

10.7　端到端示例——MyConference中的演讲者提案

前面的内容已经展示了如何在命令行中使用 Kafka，接下来我们会将其与 Node.js 应用程序进行集成，从而实现对消息的生产和消费。对于本章最后的这个示例，我们将创建一个允许演讲者向 MyConference（一家虚构的公司）提交新提案的应用程序。每个演讲者都会提交一个提案，这些提案会由 MyConference 提案团队的成员审核。审核后，演讲者就可以通过电子邮件收到有关审核结果的提醒。

10.7.1　测试数据

我们将继续使用之前章节中所使用的演讲者数据，但会添加一些元素，以便提案更加完整。例 10-10 展示了更新后的演讲提案数据。

例 10-10　data/speakerProposal.json

```
{
  "speaker": {
    "firstName": "Larson",
    "lastName": "Richard",
    "email": "larson.richard@ecratic.com",
    "bio": "Larson Richard is the CTO of ... and he founded a JavaScript meetup ..."
  },
  "session": {
    "title": "Enterprise Node",
    "abstract": "Many developers just see Node as a way to build web APIs or ...",
    "type": "How-To",
    "length": "3 hours"
  },
  "conference": {
    "name": "Ultimate JavaScript Conference by MyConference",
    "beginDate": "2017-11-06",
    "endDate": "2017-11-10"
  },
  "topic": {
    "primary": "Node.js",
    "secondary": [
      "REST",
      "Architecture",
      "JavaScript"
    ]
  },
  "audience": {
    "takeaway": "Audience members will learn how to ...",
    "jobTitles": [
      "Architects",
      "Developers"
    ],
    "level": "Intermediate"
  },
  "installation": [
    "Git",
    "Laptop",
    "Node.js"
  ]
}
```

以上示例包括了以下对象。

`speaker`
: 演讲者的联系信息。

`session`
: 有关演讲的描述，包括标题和时长。

`conference`
: 显示演讲者申请的会议。MyConference 公司同时运营多个会议，因此该对象中所包含的信息是十分重要的。

`topic`
　演讲中涉及的主要话题和次要话题。

`audience`
　目标听众的范围（初级、中级或者高级）。

`installation`
　听众在与会前应当事先做好的软件安装工作（如果需要的话）。

10.7.2　架构中的组件

以下是 MyConference 应用程序所需要的组件。

演讲提案生成程序
　使用 publish-message.sh 脚本向 `new-proposals-recvd` 主题发送 JSON 形式的演讲提案。在实际的产品开发中，这一程序的形式应该会是一个优雅的 AngularJS 应用程序，拥有不错的用户体验设计并通过 RESTful API 进行通信；不过，为了专注于介绍 JSON，我们依旧使用 publish-message.sh 这个极为简单的命令行脚本工具。

提案审核程序（即消费者）
　该程序会监听 `new-proposals-recvd` 主题，通过或拒绝某个提案，然后向 `proposals-reviewed` 主题发送相应的消息以供后续处理。在实际的企业级架构中，一般会在最前面配置 RESTful API 接口来接收提案请求，然后再将相应的消息发布到 `new-proposals-recvd` 主题中。不过与之前的生产者程序一样，此处我们也简化了示例，不使用任何 API 组件。

演讲者提醒程序（即消费者）
　该程序会监听 `proposals-reviewed` 主题，然后根据审核人员的决定生成相应的通过邮件或拒绝邮件，最后向演讲者发送提醒邮件。

电子邮件服务器（模拟）
　该服务器作为 MyConference 公司的邮件服务器来发送提醒邮件。

电子邮件客户端（模拟）
　该客户端作为演讲者的邮件客户端来接收提醒邮件。

我们将使用 MailCatcher 来模拟电子邮件的客户端和服务器端，从而简化程序的基础设施设置。

图 10-1 展示了整个应用程序中的数据流和各个组件间的交互情况。

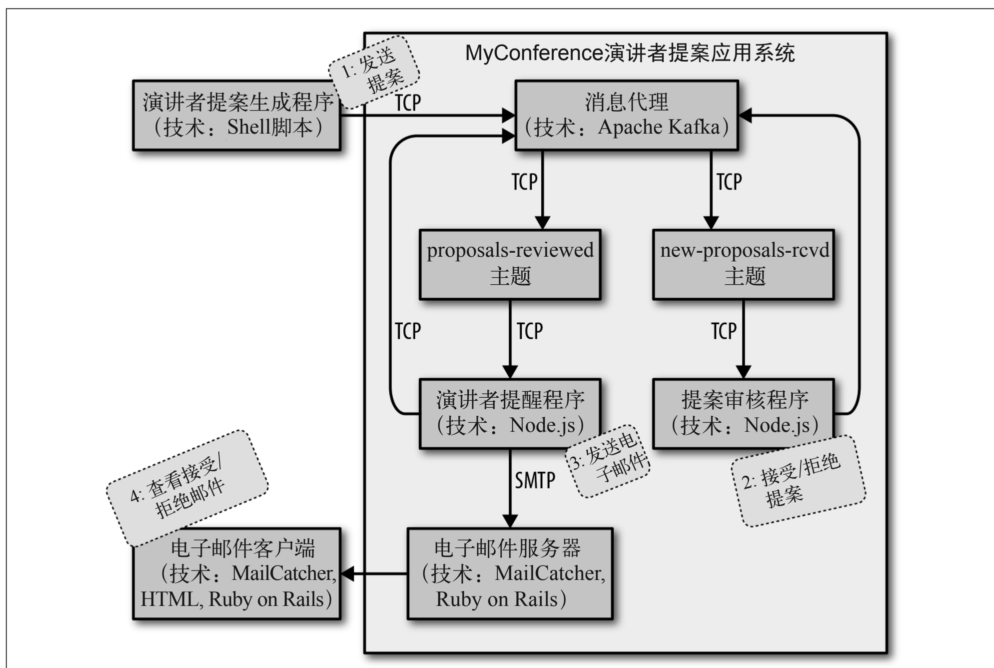


图 10-1：MyConference 演讲提案系统的架构——组件

该图表中的数据流如下所示。

(1) 在 MyConference 应用程序中，演讲者使用演讲提案生成程序向 `new-proposals-recvd` 主题发送提案消息。
(2) 提案审核程序在 `new-proposals-recvd` 主题上收到新的提案消息，做出决策，然后将相应的通过 / 拒绝消息发送到 `proposals-reviewed` 主题中。
(3) 演讲者提醒程序在 `proposals-reviewed` 主题上收到通过 / 拒绝的消息，然后创建提醒邮件并发送。
(4) 演讲者查看提醒邮件中的消息。

接下来我们将查看具体的代码，并运行这一示例。

10.7.3 配置Kafka环境

如果你运行过前面命令行的示例，那么接下来的步骤就会显得似曾相识（可以回顾之前的内容来唤起相关记忆）。我们总共需要运行 4 个命令行工具来配置本示例的环境。执行以下操作。

(1) 创建命令行环境 1。

- 启动 ZooKeeper。
- 启动 Kafka。

(2) 创建命令行环境 2。

- 创建 `proposals-reviewed` 主题。
- 创建 `new-proposals-recvd` 主题。

完成 Kafka 核心组件的配置后，我们来搭建一个电子邮件服务器，以接收通过 / 拒绝提醒邮件方面的消息。

10.7.4 配置模拟的电子邮件服务器及客户端——MailCatcher

我们将使用 MailCatcher 来完成这一功能。对于电子邮件测试来说，简单的邮件服务器是一个不错的工具，因为使用这一工具无须发送真实邮件即可完成测试。MailCatcher 具备本示例所需要的很多特性。

- 遵循标准——MailCatcher 遵循简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）。
- 安装过程简单。
- 启动 / 关闭过程简单。
- 安全特性是可选的。我知道这么说有点惊世骇俗，但事实上我们想避免对邮件服务器的用户 ID/ 密码进行配置所带来的一大堆琐事。对于我们正在开发的这种示例项目或者原型项目而言，这么做完全没问题。当然，对于比较大的原型应用程序或者真实的产品开发来说，确实应该在邮件服务器上添加安全限制。因为 MailCatcher 也可以进行用户安全配置，所以完全可以应用于更大型的示例项目。
- 优秀的 Web 用户界面，可以显示发往服务器的邮件消息。

如需了解更多有关 MailCatcher 的信息，可参考其官方网站。

如尚未安装 Ruby on Rails，可参考 A.3 节中的内容来执行相关的安装操作。在命令行中使用以下方式来安装 `mailcatcher` 包（可参考 A.3.3 节）：

```
gem install mailcatcher
```

启动 MailCatcher 服务器，你应该可以在屏幕上看到相应的结果：

```
json-at-work => mailcatcher
Starting MailCatcher
==> smtp://127.0.0.1:1025
==> http://127.0.0.1:1080
*** MailCatcher runs as a daemon by default. Go to the web interface to quit.
```

MailCatcher 以 daemon 进程的形式在后台运行，这使得用户可以在当前命令行窗口中执行其他操作。我们将在需要审核邮件时访问 MailCatcher 的 Web 用户界面（参见 10.7.9 节）。

10.7.5 配置Node.js项目环境

提案审核程序和演讲者提醒程序都是使用 Node.js 编写的。如尚未安装 Node.js，可参考 A.2 节和 A.2.5 节中的内容来执行相关的安装操作。如需依照本节的描述运行代码示例中的 Node.js 项目，可使用 `cd` 命令切换到 chapter-10/myconference 目录，并执行以下命令来安装项目依赖：

```
npm install
```

如需手动创建本节中的 Node.js 项目，可参考本书在 GitHub 上的相关指导步骤。

10.7.6 演讲提案生成程序（用于发送演讲提案）

我们会使用之前的 publish-message.sh 脚本将 speakerProposal.json 文件中的内容发送到 `new-proposals-recvd` 主题中。在命令行窗口切换到 scripts 目录下，运行以下命令：

```
./publish-message.sh -f ../data/speakerProposal.json new-proposals-recvd
```

提案审核程序会随机通过 / 拒绝提案（参见 10.7.7 节），因此你可能需要运行 3~5 次（或者更多）脚本后才能既得到通过的提醒消息，又得到拒绝的消息。

10.7.7 提案审核程序（消息的消费者和生产者）

提案审核程序会执行以下操作。

- 监听 `new-proposals-recvd` 主题来接收演讲者的提案。
- 检查提案并决定通过还是拒绝。
- 将有关提案的决定发送到 `proposals-reviewed` 主题中，以供后续处理。

myconference/proposalReviewer.js 文件中包含了所有的提案审核应用程序。例 10-11 展示了其中一部分代码，主要是关于接收 `new-proposals-recvd` 主题上的演讲提案以及相关配置方面的情况。

例 10-11　myconference/proposalReviewer.js

```
var kafka = require('kafka-node');

...

const NEW_PROPOSALS_RECEIVED_TOPIC = 'new-proposals-recvd';

...

var consumer = new kafka.ConsumerGroup({
  fromOffset: 'latest',
  autoCommit: true
}, NEW_PROPOSALS_RECEIVED_TOPIC);

// 使用接收到的JSON消息。
// 使用JSON.parse()和JSON.stringify()来处理JSON。
consumer.on('message', function(message) {
  // console.log('received kafka message', message);
  processProposal(message);
});

consumer.on('error', function(err) {
  console.log(err);
});

process.on('SIGINT', function() {
  console.log(
    'SIGINT received - Proposal Reviewer closing. ' +
    'Committing current offset on Topic: ' +
    NEW_PROPOSALS_RECEIVED_TOPIC + ' ...'
```

```
  );

  consumer.close(true, function() {
    console.log(
      'Finished committing current offset. Exiting with graceful shutdown ...'
    );

    process.exit();
  });
});
```

在以上示例中，需要注意以下几点。

- kafka-node 这一 npm 模块用于生产 / 消费 Kafka 中的消息。可以在 kafka-node 的 npm 主页及 GitHub 主页上找到更多信息。
- 监听 new-proposals-recvd 主题并使用其中的消息。
  - 实例化一个 ConsumerGroup 对象，并通过该对象使用 new-proposals-recvd 主题中的 Kafka 消息。fromOffset: 'latest' 参数表示我们需要接收该主题的最新消息，autoCommit: true 参数则表示使用每条消息后就自动提交（这样该消息会标记为"已处理"）。
  - consumer.on('message' ...) 监听消息并调用 processProposal() 函数来处理新接收到的演讲提案。后面我们会再详细介绍 processProposal() 函数。
  - consumer.on('error' ...) 在处理消息遇到报错时会将错误消息打印出来。
  - process.on('SIGINT' ...) 监听 SIGINT 信号（关闭进程），提交当前的偏移量并优雅地退出程序。
    - consumer.close(...) 提交当前的偏移量。该操作确保当前的消息被标记为已读，而程序重启后，监听相关主题的消费者程序就可以收到新的消息了。

例 10-12 展示了如何校验演讲提案并做出决定。

例 10-12　myconference/proposalReviewer.js

```
...

var fs = require('fs');
var Ajv = require('ajv');

...

const SPEAKER_PROPOSAL_SCHEMA_FILE_NAME =
  './schemas/speakerProposalSchema.json';

...

function processProposal(proposal) {
  var proposalAccepted = decideOnProposal();
  var proposalMessage = proposal.value;
  var proposalMessageObj = JSON.parse(proposalMessage);

  console.log('\n\n');
  console.log('proposalMessage = ' + proposalMessage);
  console.log('proposalMessageObj = ' + proposalMessageObj);
```

```
  console.log('Decision - proposal has been [' +
    (proposalAccepted ? 'Accepted' : 'Rejected') + ']');

  if (isSpeakerProposalValid(proposalMessageObj) && proposalAccepted) {
    acceptProposal(proposalMessageObj);
  } else {
    rejectProposal(proposalMessageObj);
  }
}


function isSpeakerProposalValid(proposalMessage) {
  var ajv = Ajv({
    allErrors: true
  });

  var speakerProposalSchemaContent = fs.readFileSync(
    SPEAKER_PROPOSAL_SCHEMA_FILE_NAME);

  var valid = ajv.validate(speakerProposalSchemaContent, proposalMessage);

  if (valid) {
    console.log('\n\nJSON Validation: Speaker proposal is valid');
  } else {
    console.log('\n\nJSON Validation: Error - Speaker proposal is invalid');
    console.log(ajv.errors + '\n');
  }

  return valid;
}

function decideOnProposal() {
  return Math.random() >= 0.5;
}

function acceptProposal(proposalMessage) {
  var acceptedProposal = {
    decision: {
      accepted: true,
      timeSlot: {
        date: "2017-11-06",
        time: "10:00"
      }
    },
    proposal: proposalMessage
  };

  var acceptedProposalMessage = JSON.stringify(acceptedProposal);
  console.log('Accepted Proposal = ' + acceptedProposalMessage);
  publishMessage(acceptedProposalMessage);
}

function rejectProposal(proposalMessage) {
  var rejectedProposal = {
    decision: {
```

```
      accepted: false
    },
    proposal: proposalMessage
  };

  var rejectedProposalMessage = JSON.stringify(rejectedProposal);
  console.log('Rejected Proposal = ' + rejectedProposalMessage);
  publishMessage(rejectedProposalMessage);
}

...
```

提案审核程序收到演讲提案的消息后，`processProposal()` 函数会执行以下操作。

- `decideOnProposal()` 会随机决定通过或者拒绝提案，从而简化逻辑。在真实的系统中，应用程序会将提案发送到某个人的工作邮箱，由人工进行审核并做出决定。
- `JSON.parse()` 会解析提案消息，确保其**语法**正确（遵循基本的 JSON 格式规则）。
- `isSpeakerProposalValid()` 函数使用 `ajv` 模块，根据 JSON Schema（schemas/speakerProposalSchema.json）校验消息。
  - 可回顾第 5 章中有关 JSON Schema 的内容。
  - 根据 JSON Schema 进行校验可以确保消息**在语义上的**正确性（消息数据具备处理提案需要的所有字段）。
  - 可以在 `ajv` 的 npm 主页和 GitHub 主页上找到更多相关信息。
- 如果提案通过，`acceptProposal()` 会执行以下操作。
  - 创建一个表示提案通过的对象，对象中包含提案通过的相关字段，以及演讲者将在会议上进行演说的具体时间信息。
  - 使用 `JSON.stringify()` 将该对象转换为 JSON。
  - 调用 `publishMessage()` 将提案通过的消息发布到 `proposals-reviewed` 主题中。
- 如果提案被拒绝（或格式不正确），`rejectProposal()` 会执行以下操作。
  - 创建一个表示拒绝提案的对象，对象中包含拒绝提案的相关字段。
  - 使用 `JSON.stringify()` 将该对象转换为 JSON。
  - 调用 `publishMessage()` 将拒绝提案的消息发布到 `proposals-reviewed` 主题中。

例 10-13 展示了如何将通过 / 拒绝的消息发布到 `proposals-reviewed` 主题中。

例 10-13　myconference/proposalReviewer.js

```
...

const PROPOSALS_REVIEWED_TOPIC = 'proposals-reviewed';

...

var producerClient = new kafka.Client(),
  producer = new kafka.HighLevelProducer(producerClient);

...

function publishMessage(message) {
  var payloads = [{
```

```
      topic: PROPOSALS_REVIEWED_TOPIC,
      messages: message
    }];

    producer.send(payloads, function(err, data) {
      console.log(data);
    });
  }

  producer.on('error', function(err) {
    console.log(err);
  });
```

这段代码使用了以下方式将消息发布到 `proposals-reviewed` 主题中。

- 实例化并使用 `HighLevelProducer` 对象将消息发布到 `proposals-reviewed` 主题中。事实上，`HighLevelProducer` 对象的实例化发生在程序文件的开头，但为方便起见，例 10-13 对其进行了展示。
- `publishMessage()` 调用 `producer.send()` 来发送消息。`producer.on('message' ...)` 则监听消息，并在收到新的演讲提案后调用 `processProposal()` 来进行处理（后文将详细介绍这一过程）。

在前面的内容中，我们只介绍了消息的生产者和消费者使用的 `kafka-node` 对象。如需了解更多详情，可访问 kafka-node 模块的文档页面来学习以下概念的相关内容：

- `HighLevelProducer`
- `ConsumerGroup`
- `Client`

探讨了提案审核程序后，现在我们创建一个新的命令行窗口，并在 myconference 路径下运行以下命令来启动该提案审核程序：

```
node proposalReviewer.js
```

当演讲提案消息发送到 `new-proposals-recvd` 主题时，你应该可以观察到提案审核程序对该消息的日志记录，以及在 `proposals-reviewed` 主题上所进行的相关决策信息：

```
json-at-work => node proposalReviewer.js



proposalMessage = { "speaker": { "firstName": "Larson", "lastName": "Richard", "email": "larson.richard@ecratic.com", "bi
o": "Larson Richard is the CTO of ... and he founded a JavaScript meetup in ..." }, "session": { "title": "Enterprise Nod
e", "abstract": "Many developers just see Node as a way to build web APIs or applications ...", "type": "How-To", "length
": "3 hours" }, "conference": { "name": "Ultimate JavaScript Conference by MyConference", "beginDate": "2017-11-06", "end
Date": "2017-11-10" }, "topic": { "primary": "Node.js", "secondary": [ "REST", "Architecture", "JavaScript" ] }, "audienc
e": { "takeaway": "Audience members will learn how to ...", "jobTitles": [ "Architects", "Developers" ], "level": "Interm
ediate" }, "installation": [ "Git", "Laptop", "Node.js" ] }
proposalMessageObj = [object Object]
Decision - proposal has been [Accepted]


JSON Validation: Speaker proposal is valid
Accepted Proposal = {"decision":{"accepted":true,"timeSlot":{"date":"2017-11-06","time":"10:00"}},"proposal":{"speaker":{
"firstName":"Larson","lastName":"Richard","email":"larson.richard@ecratic.com","bio":"Larson Richard is the CTO of ... an
d he founded a JavaScript meetup in ..."},"session":{"title":"Enterprise Node","abstract":"Many developers just see Node
as a way to build web APIs or applications ...","type":"How-To","length":"3 hours"},"conference":{"name":"Ultimate JavaSc
ript Conference by MyConference","beginDate":"2017-11-06","endDate":"2017-11-10"},"topic":{"primary":"Node.js","secondary
":["REST","Architecture","JavaScript"]},"audience":{"takeaway":"Audience members will learn how to ...","jobTitles":["Arc
hitects","Developers"],"level":"Intermediate"},"installation":["Git","Laptop","Node.js"]}}
{ 'proposals-reviewed': { '0': 12 } }
```

10.7.8　演讲者提醒程序（消息的消费者）

对提案做出通过或者拒绝的决定后，演讲者提醒程序会执行以下操作。

- 监听 `proposals-reviewed` 主题中通过 / 拒绝提案的消息。
- 格式化通过或拒绝提案的电子邮件。
- 发送通过或拒绝提案的电子邮件。

myconference/speakerNotifier.js 文件中包含了完整的演讲者提醒应用程序。例 10-14 展示了接收 `proposals-reviewed` 主题中通过 / 拒绝提案消息的部分代码。

例 10-14　myconference/speakerNotifier.js

```
var kafka = require('kafka-node');

...

const PROPOSALS_REVIEWED_TOPIC = 'proposals-reviewed';

...

var consumer = new kafka.ConsumerGroup({
  fromOffset: 'latest',
  autoCommit: true
}, PROPOSALS_REVIEWED_TOPIC);

...

consumer.on('message', function(message) {
  // console.log('received message', message);
  notifySpeaker(message.value);
});

consumer.on('error', function(err) {
  console.log(err);
});

process.on('SIGINT', function() {
  console.log(
   'SIGINT received - Proposal Reviewer closing. ' +
   'Committing current offset on Topic: ' +
    PROPOSALS_REVIEWED_TOPIC + ' ...'
  );

  consumer.close(true, function() {
    console.log(
      'Finished committing current offset. Exiting with graceful shutdown ...'
    );

    process.exit();
  });
});

...
```

演讲者提醒程序使用以下方式监听并使用 `proposals-reviewed` 主题中的消息。

- 实例化并利用 `ConsumerGroup` 对象来使用 `proposals-reviewed` 主题中的 Kafka 消息。提醒程序的初始配置工作与提案审核程序的代码类似。
- `consumer.on('message' ...)` 监听消息，并在收到新的提案审核结果后调用 `notifySpeaker()` 来进行处理（后文将详细介绍这一过程）。
- `consumer.on('error' ...)` 和 `process.on('SIGINT' ...)` 函数的作用与提案审核程序中的相同。

例 10-15 展示了如何处理提案的通过 / 拒绝消息，并使用 Handlebars 将结果格式化为相应的电子邮件（详见第 7 章）。

例 10-15　myconference/speakerNotifier.js

```
...

var handlebars = require('handlebars');
var fs = require('fs');

...

const EMAIL_FROM = 'proposals@myconference.com';
const ACCEPTED_PROPOSAL_HB_TEMPLATE_FILE_NAME =
  './templates/acceptedProposal.hbs';

const REJECTED_PROPOSAL_HB_TEMPLATE_FILE_NAME =
  './templates/rejectedProposal.hbs';

const UTF_8 = 'utf8';

...

function notifySpeaker(notification) {
  var notificationMessage = createNotificationMessage(notification);

  sendEmail(notificationMessage);
}

function createNotificationMessage(notification) {
  var notificationAsObj = JSON.parse(notification);
  var proposal = notificationAsObj.proposal;

  console.log('Notification Message = ' + notification);
  var mailOptions = {
    from: EMAIL_FROM, // 发送者地址
    to: proposal.speaker.email, // 接受者列表
    subject: proposal.conference.name + ' - ' + proposal.session.title, // 主题
    html: createEmailBody(notificationAsObj)
  };

  return mailOptions;
}

function createEmailBody(notification) {
```

```
  // 阅读Handlebars模板文件。
  var hbTemplateContent = fs.readFileSync(((notification.decision.accepted) ?
    ACCEPTED_PROPOSAL_HB_TEMPLATE_FILE_NAME :
    REJECTED_PROPOSAL_HB_TEMPLATE_FILE_NAME), UTF_8);

  // 将模板编译为一个方程。
  var template = handlebars.compile(hbTemplateContent);
  var body = template(notification); // 渲染模板。

  console.log('Email body = ' + body);
  return body;
}

...
```

提醒程序收到通过 / 拒绝提案的消息后，`notifySpeaker()` 函数会执行以下操作。

- 调用 `createNotificationMessage()` 来创建发送给演讲者的提醒邮件。
  - 使用 `JSON.parse()` 将通过 / 拒绝提案的消息解析为对象。
  - 调用 `createEmailBody()`。
    - ♦ 根据上述解析后的对象，使用 `handlebars` 模块来生成 HTML 格式的电子邮件消息。
    - ♦ 可参考第 7 章来回顾有关 Handlebars 的内容。
    - ♦ 如需了解更多信息，可参考 `handlebars` 的 npm 主页与 GitHub 主页。
- 调用 `sendEmail()` 将提醒邮件发送给演讲者（参见以下示例）。

例 10-16 展示了发送通过 / 拒绝邮件的过程。

例 10-16　myconference/speakerNotifier.js

```
...

var nodeMailer = require('nodemailer');

...

const MAILCATCHER_SMTP_HOST = 'localhost';
const MAILCATCHER_SMTP_PORT = 1025;

var transporter = nodeMailer.createTransport(mailCatcherSmtpConfig);

...

function sendEmail(mailOptions) {
  // 通过已定义的运输对象发送邮件
  transporter.sendMail(mailOptions, function(error, info) {
    if (error) {
      console.log(error);
    } else {
      console.log('Email Message sent: ' + info.response);
    }
  });
}
```

演讲者提醒程序使用以下方式将邮件消息发送到 MailCatcher 服务器。

- 实例化并使用 `nodemailer` 传输对象来发送电子邮件。`MAILCATCHER_SMTP_...` 常量用于保存本机所启动的 MailCatcher 服务器的地址与端口。`nodemailer` 传输对象的实例化实际发生在文件开头，出于方便的目的，上述示例对其进行了展示。
- `sendEmail()` 函数调用 `transporter.sendMail()` 来发送邮件消息。
- `nodemailer` 是一个使用 SMTP 来发送邮件消息的通用 npm 模块。如需了解更多信息，可参考其 npm 主页及社区主页。

现在我们打开一个新的命令行窗口，在 myconference 路径下运行以下命令来启动演讲者提醒程序：

```
node speakerNotifier.js
```

当 `proposals-reviewed` 主题收到通过 / 拒绝提案的消息时，你应该可以观察到演讲者提醒程序对该消息的日志记录，以及其发送的提醒邮件：

```
json-at-work => node speakerNotifier.js
Notification Message = {"decision":{"accepted":true,"timeSlot":{"date":"2017-11-06","time":"10:00"}},"proposal":{"speaker
":{"firstName":"Larson","lastName":"Richard","email":"larson.richard@ecratic.com","bio":"Larson Richard is the CTO of ...
 and he founded a JavaScript meetup in ..."},"session":{"title":"Enterprise Node","abstract":"Many developers just see No
de as a way to build web APIs or applications ...","type":"How-To","length":"3 hours"},"conference":{"name":"Ultimate Jav
aScript Conference by MyConference","beginDate":"2017-11-06","endDate":"2017-11-10"},"topic":{"primary":"Node.js","second
ary":["REST","Architecture","JavaScript"]},"audience":{"takeaway":"Audience members will learn how to ...","jobTitles":["
Architects","Developers"],"level":"Intermediate"},"installation":["Git","Laptop","Node.js"]}}
Email body = <!DOCTYPE html>
<html>
  <body>
     <p>
     Larson,
     </p>
     <p>
     We are pleased to inform you that your talk on <u>Enterprise Node</u>
     has been accepted for the <b>Ultimate JavaScript Conference by MyConference</b>.
     </p>
     <p>
     Your session scheduled for 2017-11-06 at 10:00.
     </p>
     <p>
     Sincerely,<br/>
     The Ultimate JavaScript Conference by MyConference Event Team.
     </p>
  </body>
</html>
Email Message sent: 250 Message accepted
```

10.7.9 用MailCatcher实现审核结果的电子邮件提醒功能

最后，我们查看一下由提醒程序所生成并发送到演讲者的提醒消息。

在本机访问 http://localhost:1080 即可看到 MailCatcher 的用户界面。图 10-2 展示了概要页面，其中列举了 MyConference 应用程序通过 Handlebars 生成的电子邮件消息。

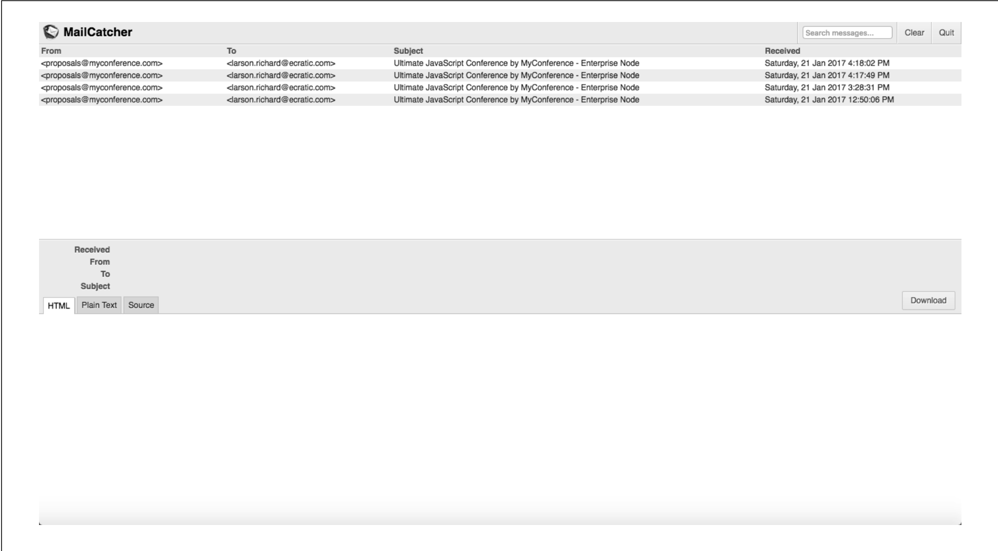


图 10-2：MailCatcher 中的演讲者提醒消息

点击其中一些邮件，有些邮件显示提案审核通过，如图 10-3 所示。

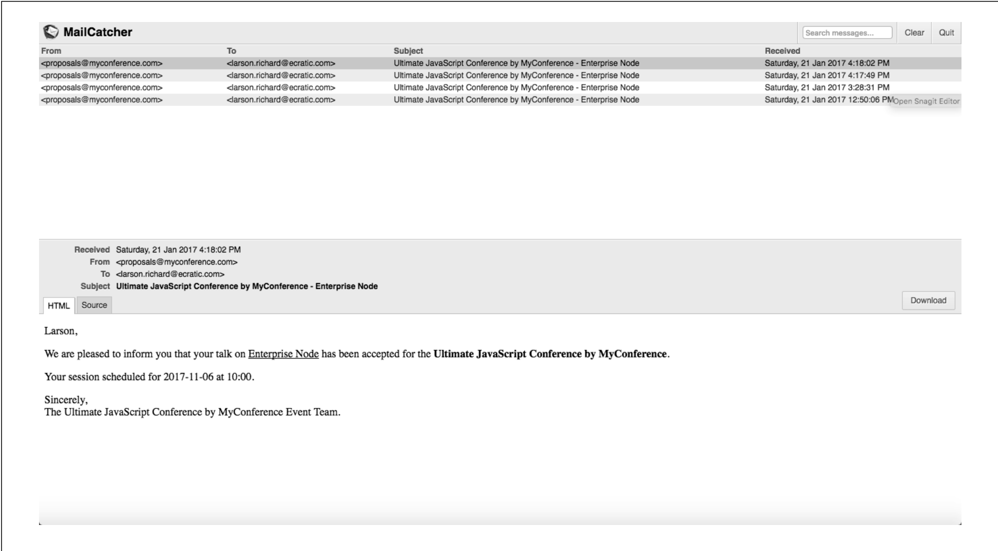
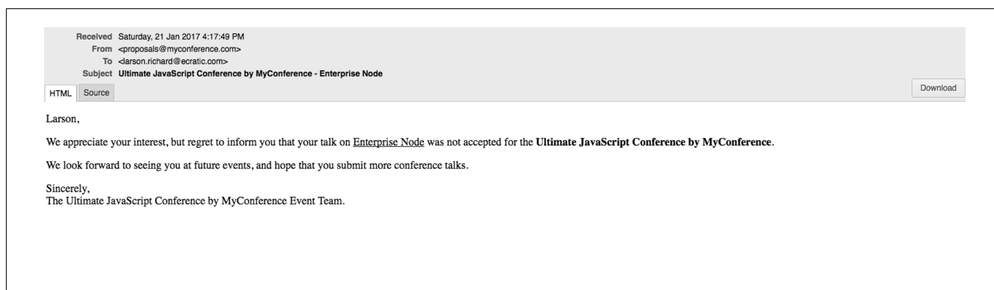


图 10-3：MailCatcher 中的演讲提案通过消息

图 10-4 则显示了一封拒绝提案的邮件。



Received Saturday, 21 Jan 2017 4:17:49 PM
From <proposals@myconference.com>
To <larson.richard@ecratic.com>
Subject **Ultimate JavaScript Conference by MyConference - Enterprise Node**

HTML Source Download

Larson,

We appreciate your interest, but regret to inform you that your talk on Enterprise Node was not accepted for the **Ultimate JavaScript Conference by MyConference**.

We look forward to seeing you at future events, and hope that you submit more conference talks.

Sincerely,
The Ultimate JavaScript Conference by MyConference Event Team.

图 10-4：MailCatcher 中拒绝演讲提案的消息

MailCatcher 的 Web 用户界面的工作机制如下。

- 可以点击 Download 按钮来下载当前的电子邮件。该操作会将邮件消息保存为 EML 格式的文件（后缀名为 .eml），该格式：
  - 遵循了 MIME 822 标准；
  - 与以下邮件客户端兼容：MS Outlook、Outlook Express、Apple Mail、Mozilla Thunderbird 等；
  - 保留了原始的 HTML 格式及邮件头。
- 可以点击页面右上角的 Quit 按钮来关闭 MailCatcher 后台进程。

10.8 本章回顾

本章介绍了以下内容。

- 在命令行中使用 Kafka 来生成 / 使用 JSON 消息。
- 设计并实现一个小型的端到端 MyConference 示例应用程序，该应用程序采用 Kafka 主题、Node.js 和模拟的电子邮件服务器来处理基于 JSON 的演讲者应用程序。

附录A

安装指南

该附录将介绍与本书代码示例有关的软件安装与配置。

A.1 在浏览器中安装JSON工具

这部分内容将介绍如何在浏览器中安装 JSON 工具。

A.1.1 在Chrome和Firefox中安装JSONView

JSONView 可以在 Chrome 和 Firefox 中优化显示 JSON。你可以参考 JSONView 网站上的安装步骤在自己的浏览器上进行相应的安装。

A.1.2 JSONLint

可以使用 JSONLint 来在线校验 JSON 文档。JSONLint 无须安装。

A.1.3 JSON Editor Online

可以使用 JSON Editor Online 对 JSON 文档进行建模。作为 Web 应用程序，JSON Editor Online 无须安装即可使用。

A.1.4 安装Postman

可以使用 Postman 对 RESTful API 进行完整的测试。Postman 可以发送 HTTP `GET`、`POST`、`PUT` 和 `DELETE` 请求，也可以设置 HTTP 头部。你可以 Chrome 插件的形式来安装 Postman，也可将其安装为 macOS、Linux 或 Windows 上的独立 GUI 应用程序。关于安装步骤，可参考其官方网站。

A.2 安装Node.js

本书使用的 Node.js 版本为 `v6.10.2`，即撰写本书时的最新稳定版。

A.2.1 用NVM在macOS和Linux上安装Node.js

虽然可以在 Node.js 的官方网站上找到安装包并进行安装，但使用这种安装方式后不太容易更换 Node.js 的版本。因此，我们使用 NVM（Node Version Manager，Node 版本管理器）来完成 Node.js 的安装工作。NVM 使得 Node.js 的安装、卸载和版本升级变得更加简单。

1. 安装配置NVM

首先，使用以下两种方式之一来安装 NVM：

- 安装脚本；
- 手动安装。

然后，确保 NVM 可以正常运行。在命令行中对其执行 source 操作：

```
source ~/.nvm/nvm.sh
```

操作完成后，NVM 即可在接下来的安装过程中正常工作。

如果运行的命令行环境是 bash，则可通过以下操作使得命令行对 NVM 进行自动配置。

- 在 $HOME/.bashrc 文件中添加以下代码。

```
source ~/.nvm/nvm.sh export NVM_HOME=~/.nvm/v6.10.2
```

- 在 $HOME/.bashrc_profile 文件中添加以下代码。

```
[[ -s $HOME/.nvm/nvm.sh ]] && . $HOME/.nvm/nvm.sh # This loads NVM
```

值得注意的是，Bourne Shell 和 Korn Shell 中的有些步骤非常类似。

2. 用NVM安装Node.js

安装好 NVM 后，即可用其来安装 Node.js。

(1) 运行 `nvm ls-remote` 来查看可安装的远端（不在本机上）Node.js 版本。
(2) 使用以下命令来安装 `v6.10.2` 版本的 Node.js。

```
nvm install v6.10.2
```

- 所有版本的 Node.js 都会安装到 `$HOME/.nvm` 中。

(3) 在所有的新命令行环境中设置默认的 Node.js 版本。

```
nvm alias default v6.10.2
```

- 如果忽略这步操作，则退出当前命令行环境后 `node` 和 `npm` 命令将不再生效。
- 操作完成后，退出当前命令行环境。

在新的命令行环境中，将 npm 升级至最新版本。

```
npm update -g npm
```

然后，执行以下的检查确认工作：

- `nvm ls`，可以观察到 `... -> v6.10.2 system default -> v6.10.2`；
- `node -v`，返回结果为 `v6.10.2`；
- `npm -v`，返回结果类似 `4.6.1`。

如需查看 NVM 的所有功能，可使用 `nvm --help`。

如果使用 Node.js 的“读取—求值—输出—循环”这个交互式编程环境，则可以观察到以下结果：

```
json-at-work => node
-> .exit
```

3. 避免使用sudo

npm 命令可能会要求使用 `sudo` 权限，但这会造成繁琐与不便。同时，给予 npm 命令 `sudo` 权限也会带来安全隐患，因为 npm 会用 root 权限来运行可执行脚本。为了避免这一点，可执行以下操作：

```
sudo chown -R $USER ~/.nvm
```

如果你的 Node.js 是用 NVM 安装的（这意味着所有的 Node.js 安装都会在 ~/.nvm 路径下进行），则上述操作可以成功避免对 sudo 的使用。这一技巧源于 Isaac Z. Schlueter 在 How to Node 网站中撰写的一篇文章。

4. 优化REPL——mynode

从使用者的角度来看，REPL 的默认行为有值得改进之处。默认情况下，敲击回车键后，对于大多数 JavaScript 语句，REPL 都会显示 `undefined`。这一行为的原因在于：JavaScript 中的函数总是会返回一些值，而在没有显式声明返回值的情况下会默认返回 `undefined`。这一行为低效而又不便。以下是其中一个示例：

```
json-at-work => node
-> Hit Enter
-> undefined
-> var y = 5
-> undefined
-> .exit
```

可以在 .bashrc 文件（或者 Bourne/Korn Shell 中的其他启动配置文件）中添加以下代码来关闭 REPL 中的 `undefined` 显示。

```
source ~/.nvm/nvm.sh

...

alias mynode="node -e \"require('repl').start({ignoreUndefined: true})\""
```

添加代码后，退出当前命令行环境，然后再重新打开一个命令行窗口。像这样定义一个新的别名（即 `mynode`）比重新定义 `node` 要更安全。在这种方式下，`node` 依旧可以在命令行中正常工作，并成功运行 JavaScript 文件。与此同时，可以使用 `mynode` 作为新的 REPL 命令：

```
json-at-work => mynode
-> var x = 5
-> .exit
```

至此，Node.js 中的 REPL 就可以如预期般工作了，因为屏幕上不会再显示繁琐不便的 `undefined`。

A.2.2 在Windows上安装Node.js

多亏了 Corey Butler 的 `nvm-windows` 应用程序，NVM 也可以在 Windows 上正常工作。`nvm-windows` 是 `nvm` 在 Windows 环境中的移植版本。我可以在 Windows 7 环境中成功使用 `nvm-windows`。

用nvm-windows在Windows上安装Node.js

以下是具体的安装步骤。

(1) 访问 `nvm-windows` 的下载页面。
(2) 将最新的 nvm-setup.zip 文件下载到本机的 Downloads 文件夹。
(3) 使用自己喜欢的工具来解压 nvm-setup.zip 文件。
(4) 运行 nvm-setup.exe 来启动安装向导。使用向导中的默认设置并接受软件的 MIT 许可证。
  a. 下载地址为 C:\Users{username}\AppData\Roaming\nvm。
  b. 安装完成后，点击 Finish 按钮。
  c. 安装过程中会在 Windows 机器上配置运行 Node.js 所必需的环境变量。
(5) 确保 `PATH` 环境变量中包含 NVM。
  a. 打开控制面板→系统→高级系统设置。
  b. 在高级系统设置窗口中点击“环境变量”。
  c. 之前的安装过程应该已经将 `NVM_HOME` 添加到环境变量中，其值为 C:\Users{username}\AppData\Roaming\nvm。
  d. `NVM_SYMLINK` 环境变量应当指向 C:\Program Files\nodejs。
  e. `PATH` 环境变量中应当包括 `NVM_HOME` 和 `NVM_SYMLINK`。
(6) 用 `nvm-windows` 安装 Node.js。
  a. 运行 `nvm list available` 来查看可安装的版本。
  b. 运行 `nvm install v6.10.2`。
  c. 设置 Node.js 的版本：`nvm use v6.10.2`。
  d. 测试刚安装的 Node：`node -v`。

A.2.3 卸载Node.js

如果当前机器上安装过 Node.js，但已经无法正常工作，那么你可能需要将其完全卸载。需要卸载的可执行文件包括 `node` 和 `npm`。

1. 在macOS上卸载Node.js

卸载工作比较复杂，本书中的卸载操作源自 Clay 在 Hungred Dot Com 网站上所发表的文章。如果 Node.js 是使用 Homebrew 来安装的，则在命令行中运行 `brew uninstall node` 即可。

如果 Node.js 不是使用 Homebrew 来安装的，则可执行以下操作。

- 用 cd 命令切换到 /usr/local/lib 路径，删除所有的 node 可执行文件和 node_modules 文件夹。
- 用 cd 命令切换到 /usr/local/include 路径，删除所有的 node 可执行文件和 node_modules 文件夹。
- 用 cd 命令切换到 /usr/local/bin 路径，删除所有的 node 可执行文件。

除此之外，你可能还需要执行以下操作：

```
rm -rf /usr/local/bin/npm
rm -rf /usr/local/share/man/man1/node.1
rm -rf /usr/local/lib/dtrace/node.d
rm -rf $USER/.npm
```

2. 在Linux上卸载Node.js

本书中在 Linux 上卸载 Node.js 的方法源自 Stack Overflow 和 GitHub 上的相关指导。具体操作如下所示。

(1) 使用 which node 来查看 node 的安装路径。假设该路径为 /usr/local/bin/node。
(2) 用 cd 命令切换到 /usr/local 目录。
(3) 执行以下命令。

```
sudo rm -rf bin/node
sudo rm -rf bin/npm
sudo rm -rf lib/node_modules/npm
sudo rm -rf lib/node
sudo rm -rf share/man/*/node.*
```

3. 在Windows上卸载Node.js

本书中在 Windows 上卸载 Node.js 的方法源自 Team Treehouse 网站中的一篇文章。以下是具体的操作步骤。

(1) 打开 Windows 的控制面板。
(2) 选择“程序和功能”。
(3) 点击“卸载程序”。
(4) 选择 Node.js，然后点击卸载链接。

A.2.4　安装Yeoman

Yeoman 由以下部分组成：

- yo，用于快速搭建项目；
- npm 或者 bower，用于包管理；
- gulp 或者 grunt，用于构建系统。

在本书的代码示例中，gulp 和 grunt-cli 会用于构建系统。虽然本书的主要构建工具是 gulp，但有时也需要 grunt-cli 来执行一些 gulp 任务。

我选择 bower 作为包管理工具。

以下是具体的安装步骤。

- 安装 yo：
  - npm install -g yo
  - 测试 yo 的安装：yo --version
- 安装 bower：
  - npm install -g bower
  - 测试 bower 的安装：bower --version
- 安装 gulp：
  - npm install -g gulp-cli
  - 测试 gulp 的安装：gulp --version
- 安装 grunt-cli：
  - npm install -g grunt-cli
  - 测试 grunt-cli 的安装：grunt --version

如需了解更多信息，可参考 Yeoman 的安装配置页面。

安装Yeoman生成器generator-webapp

参考 generator-webapp 的 GitHub 主页，使用以下方式来安装生成器：

```
npm install -g generator-webapp
```

A.2.5　安装npm模块

本书会在命令行中使用到以下 npm 模块，因此我们需要对它们进行全局安装：

- jsonlint
- json
- ujs-jsonvalidate
- http-server
- json-server
- jq-tutorial

1. 安装jsonlint

该模块是 JSONLint 网站所对应的 npm 包，用于校验 JSON 文档。具体信息可参考其 GitHub 主页。

安装 jsonlint：

```
npm install -g jsonlint
```

校验 JSON 文档：

```
jsonlint basic.json
```

2. 安装json

json 模块提供了在命令行中处理 JSON 文档（如优化显示）的功能。json 与 jq 类似，但功能没有 jq 强大。

安装 `json`：

```
npm install -g json
```

如需了解 `json` 的使用信息，可参考其 GitHub 主页。`json` 能够以 npm 包的形式来使用。

3. 安装`ujs-jsonvalidate`

该模块是 JSON Validate 网站所对应的 npm 包，用于根据 JSON Schema 来校验 JSON 文档。具体信息可参考其 Github 主页。

安装 `ujs-jsonvalidate`：

```
npm install -g ujs-jsonvalidate
```

校验 JSON 文档：

```
validate basic.json basic-schema.json
```

4. 安装`http-server`

`http-server` 是一个简单的 Web 服务器，用于将本机系统中当前路径下的文件以静态内容资源的形式向外暴露。因为 `http-server` 文档齐全，命令行选项与关闭方式也完全符合直觉，所以我很喜欢使用该工具。具体信息可参考其 GitHub 主页与 npm 主页。

安装 `http-server`：

```
npm install -g http-server
```

可以使用以下方式来运行 Web 服务器：

```
http-server -p 8081
```

可以使用以下地址来访问文件资源：

```
http://localhost:8081
```

敲击 Ctrl-C 即可关闭服务器。

5. 安装`json-server`

`json-server` 是一个模拟的 REST 服务器，可接受 JSON 文件并将其暴露为 RESTful 服务。具体信息可参考其 GitHub 主页。

安装 `json-server`：

```
npm install -g json-server
```

可以使用以下方式来运行 `json-server`：

```
json-server -p 5000 ./speakers.json
```

可以使用以下地址来访问 RESTful 资源：

```
http://localhost:5000/speakers
```

6. 安装Crest

Crest 是一个小型的 REST 服务器，为 MongoDB 数据库提供了 RESTful 的封装层。具体信

息可参考其 GitHub 主页。安装 Crest 的最简方法本来应该是通过 npm 全局安装，可惜这条路目前走不通。因此，需要采用 git clone 的方案，具体操作如下。

(1) 用 cd 命令将当前路径切换到其他开发项目所在的目录下。姑且称此目录为 projects。

```
cd projects
```

(2) 克隆 Crest 的 Git 仓库。

```
git clone git://github.com/Cordazar/crest.git
```

(3) 切换到 crest 目录。

```
cd crest
```

(4) 编辑 config.json 文件，删除其中的 username 和 password 部分信息。当然，这么做是不安全的，你稍后可以再加上这些字段信息，并赋以正确的值，只要确保其与 MongoDB 中的密码一致即可。就目前而言，我们只想快速上手，因此 config.json 文件会如下所示。

```
{
  "db": { "port": 27017, "host": "localhost" },
  "server": { "port": 3500, "address": "0.0.0.0" },
  "flavor": "normal",
  "debug": true
}
```

(5) 确保已经安装并启动了 MongoDB。

(6) 在一个新的命令行窗口中，用 node server 命令来启动 Crest。你应该可以观察到以下结果。

```
node server

DEBUG: util.js is loaded
DEBUG: rest.js is loaded
crest listening at http://:::3500
```

7. 安装jq-tutorial

jq-tutorial 模块可以在命令行中提供不错的 jq 教程，可以用以下方式对其进行安装：

```
npm install -g jq-tutorial
```

安装后即可在命令行中运行该教程：

```
jq-tutorial
```

A.3 安装Ruby on Rails

安装 Ruby on Rails 的方法有很多种：

- Rails 安装程序；
- ruby-install；

- Ruby 版本管理器（Ruby Version Manager，RVM）+ `rails` 包；
- `rbenv` + `rails` 包。

A.3.1　在macOS和Linux上安装Rails

我喜欢在 macOS 和 Linux 上使用 RVM 来安装 Rails，因为这种方式在升级切换 Ruby 版本时非常简单。可访问 RVM 的官方网站，并参考网站上的安装步骤来安装 RVM。

可以使用 RVM 按照以下步骤来安装 Ruby。

(1) 查看可安装的 Ruby 版本。

```
rvm list known
```

(2) 用以下方式安装 Ruby `v2.4.0`。

```
rvm install 2.4.0
```

(3) 检查已安装的 Ruby 版本，应该可以观察到以下结果。

```
ruby -v
ruby 2.4.0
```

(4) 安装好 Ruby 后，就可以使用以下方式来安装 Rails 了。

```
gem install rails
```

(5) 检查已安装的 Rails 版本，应该可以观察到以下结果。

```
rails -v
Rails Rails 5.0.2
```

至此，安装工作就算完成了。

可以通过以下步骤轻松地将 Ruby 和 Rails 升级到新版本。

(1) 安装新版本的 Ruby（以 2.*x* 为例）。

```
rvm install 2.x
```

(2) 使用新版本的 Ruby。

```
rvm use 2.x
```

(3) 与之前一样，安装 `rails` gem。

A.3.2　在Windows上安装Rails

可以在 Windows 环境下使用 Rails 安装程序来安装 Rails，具体操作如下。

- 下载 Windows 中的安装程序。
- 运行安装程序并使用程序中的默认选项。

我在 Windows 7 环境下成功使用 Rails 安装程序完成了安装工作。Rails 安装程序的官方网页中包含了非常优秀的 RoR 教程，以及安装过程中可能出现的问题的解决方案。

A.3.3　安装Ruby gem

除 Rails 外，本书还使用到了以下 Ruby gem，因此我们会以全局方式对其进行安装：

- `multijson`
- `oj`
- `awesome_print`
- `activesupport`
- `minitest`
- `mailcatcher`

1. 安装multi_json

`multi_json` 封装了 gem 的选择和调用，自动选择当前应用程序环境中最快的 JSON 包。具体的安装过程如下所示：

```
gem install multi_json
```

2. 安装oj

很多人认为 Optimized JSON（`oj`）是 Ruby 中最快的 JSON 处理工具。具体的安装过程如下所示：

```
gem install oj
```

3. 安装awesome_print

`awesome_print` 能够以优化显示的方式来打印 Ruby 对象，可用于调试过程。具体的安装过程如下所示：

```
gem install awesome_print
```

4. 安装activesupport

`activesupport` 提供了从 Rails 中抽取出来的一些功能，其 JSON 模块提供了在驼峰式命名和下划线分隔命名间进行转换的功能。具体的安装过程如下所示：

```
gem install activesupport
```

5. 安装mailcatcher

`mailcatcher` 是一个简单的邮件（SMTP）服务器。使用该优秀工具的话，无须发送真正的电子邮件即可完成对邮件功能的测试。具体的安装过程如下所示：

```
gem install mailcatcher
```

A.4　安装MongoDB

可参考 MongoDB 官方网站上的安装文档，根据相应的步骤在本机上安装并运行 MongoDB。

A.5　安装Java环境

本书中的 Java 环境包括以下两部分内容：

- Java SE
- Gradle

A.5.1　安装Java SE

本书使用 Java 标准版本（Standard Edition, SE）8，因此可访问 Oracle 网站上 Java SE 8 的下载页面。

还可以在该页面上看到 JDK（Java Developer Kit，Java 开发者套件）这一术语。JDK 是 Java SE 的旧称。只需要找到 Java SE Development Kit，接受许可证，并将文件下载到本机操作系统中即可。下载并运行安装程序后，需要在操作系统中配置 Java 命令行环境。

执行完操作系统需要的配置工作后，运行以下命令：

```
java -version
```

可以观察到类似以下的结果：

```
java version "1.8.0_72"
Java(TM) SE Runtime Environment (build 1.8.0_72-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.72-b15, mixed mode)
```

1. 在macOS上配置Java

在 .bashrc 文件中使用以下代码来配置 `JAVA_HOME` 环境变量，并将其添加到 `PATH` 中：

```
...

export
JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.x.y.jdk/Contents/Home
# x和y分别为小版本号和补丁版本号

...

export PATH=...:$\{JAVA_HOME}/bin:...
```

2. 在Linux上配置Java

在 .bashrc 文件中使用以下代码来配置 `JAVA_HOME` 环境变量，并将其添加到 `PATH` 中：

```
...

export JAVA_HOME=/usr/java/jdk1.x.y/bin/java # x和y分别为小版本号和补丁版本号

...

export PATH=...:$\{JAVA_HOME}/bin:...
```

然后更新环境变量：

```
source ~/.bashrc
```

上述内容源于 nixCraft 上的相关文章。

3. 在Windows上配置Java

Java 的 Windows 安装程序一般会将 JDK 安装到 C:\Program Files\Java 或者 C:\Program Files (x86)\Java 中。

安装完成后，执行以下配置操作。

(1) 在桌面上右键点击“我的电脑”，然后选择“属性”。
(2) 点击“高级”选项。
(3) 点击“环境变量”按钮。
(4) 在“系统变量”下方点击“新建”。
(5) 输入变量名为 `JAVA_HOME`。
(6) 输入变量值为刚安装的 Java Development Kit 的路径（JDK 安装目录）。
(7) 点击“确定”。
(8) 点击“应用”。

上述内容源于 Robert Sindall 的一篇博客文章。

A.5.2　安装Gradle

Gradle 用于构建源代码，并测试代码。可访问 Gradle 的安装指南，根据文档上的步骤在本机操作系统中对其进行安装。安装完成后，在命令行中运行 `gradle -v`，应该可以观察到类似以下的结果：

```
gradle -v

------------------------------------------------------------
Gradle 3.4.1
------------------------------------------------------------
```

我成功地在 macOS 上使用过 Homebrew 来安装 Gradle。

A.6　安装jq

`jq` 是一个命令行中的 JSON 处理工具。可根据 `jq` 在 GitHub 主页上的下载指南文档来安装 `jq`。

`jq` 的运行依赖于 cURL。

A.7　安装cURL

cURL 可以实现包括 HTTP 在内的多种协议间的通信。可以使用 cURL 在命令行中向 RESTful API 发起 HTTP 调用。

A.7.1　在macOS上安装cURL

与 Linux 一样，你的 Mac 机器中可能已经默认安装了 cURL。可以使用以下方式来检查：

```
curl --version
```

如果已经安装 cURL，那么你就无须再做任何其他操作。否则就需要手工来安装 cURL。在 macOS 上，我将 Homebrew 作为包安装管理工具，因此可以使用以下命令来安装 cURL：

```
brew install curl
```

A.7.2　在Linux上安装cURL

可以使用以下命令来检查 cURL 当前的安装情况：

```
curl --version
```

如尚未安装，则可在命令行中执行以下命令：

```
sudo apt-get install curl
```

该命令可以成功地在 Ubuntu 或者 Debian 上安装 cURL。

A.7.3　在Windows上安装cURL

可以通过以下步骤在 Windows 上安装 cURL。

(1) 访问 cURL 的下载页面。
(2) 选择包类型为：curl 可执行文件。
(3) 选择操作系统为：Windows/Win32 或者 Win64。
(4) 选择用户喜好为：`Cygwin`（如果使用了 Cygwin）或者 `Generic`（如果未使用 Cygwin）。
(5) 选择 Win32 的版本（如果第 3 步中选择了 Windows/Win32）为：Unspecified。

上述内容源自 Stack Overflow 上的相关讨论。

A.8　安装Apache Kafka

第 10 章使用了 Apache Kafka 来实现 JSON 消息系统。Kafka 的运行依赖于 Apache ZooKeeper，因此你还需要安装 ZooKeeper。因为 Kafka 是用 Java 开发的，所以在继续深入前，需要先确保本机上已经安装好 Java 环境。

A.8.1　在macOS上安装Kafka

在 macOS 上安装 Kafka 的最简方式是使用 Homebrew。可在命令行中运行以下命令：

```
brew install kafka
```

该操作会在机器上安装 Kafka 和 ZooKeeper。

A.8.2　在UNIX上安装Kafka

可以通过以下方式来安装 ZooKeeper。

- 从 ZooKeeper 的发布页面中下载 ZooKeeper。
- 解压下载的最新 ZooKeeper 文件。

```
tar -zxf ZooKeeper-3.4.9.tar.gz
```

- 在 ~/.bashrc 文件中添加系统环境变量。

```
export ZooKeeper_HOME = <Zookeeper-Install-Path>/zookeeper-3.4.9
export PATH=$PATH:$ZOOKEEPER_HOME/bin
```

可以通过以下方式来安装 Kafka。

(1) 从 Kafka 的下载页面中下载 Kafka。
(2) 解压下载的最新 Kafka 文件。

```
tar -zxf  kafka_2.11-0.10.1.1.tgz
```

(3) 在 ~/.bashrc 文件中添加系统环境变量。

```
export KAFKA_HOME = <Kafka-Install-Path>/zookeeper-3.4.9
export PATH=$PATH:$KAFKA_HOME/bin
```

上述内容源自 TutorialsPoint 网站上的相关文章。

A.8.3 在Windows上安装Kafka

可以通过以下方式来安装 ZooKeeper。

(1) 从 ZooKeeper 的下载页面中下载 ZooKeeper。
(2) 使用喜欢的解压工具将 ZooKeeper 解压到 C 盘。
(3) 使用以下方式添加系统环境变量。
  a. 在 Windows 中，打开控制面板→系统→高级系统设置→环境变量。
  b. 对于最新下载的 ZooKeeper，创建新的系统环境变量。

```
ZOOKEEPER_HOME = C:\zookeeper-3.4.9
```

  c. 将 ZooKeeper 添加到 `PATH` 环境变量中，具体操作方式为：编辑 `PATH` 并在其最后添加以下后缀。

```
;%ZOOKEEPER_HOME%\bin;
```

可以通过以下方式来安装 Kafka。

(1) 从 Kafka 的下载页面中下载 Kafka。
(2) 使用喜欢的解压工具将 Kafka 解压到 C 盘。
(3) 使用以下方式添加系统环境变量。
  a. 在 Windows 中，打开控制面板→系统→高级系统设置→环境变量。
  b. 对于最新下载的 Kafka，创建新的系统环境变量。

```
KAFKA_HOME = C:\kafka_2.11-0.10.1.1
```

  c. 将 Kafka 添加到 `PATH` 环境变量中，具体操作方式为：编辑 `PATH` 并在其最后添加以下后缀。

```
;%KAFKA_HOME%\bin;
```

上述内容源自 DZone 网站上的相关文章。

A.9 内容参考

本附录的 AsciiDoc 版本是由 Pandoc 根据本书 GitHub 上的 Markdown 文章所生成的。

附录B

JSON社区

JSON 社区非常活跃。你可以访问以下群组、参与社区并进一步学习。

JSON.org
: Douglas Crockford 创立的 JSON 网站，即 JSON 的起源。

JSON Yahoo! 群组
: 该 Yahoo! 群组从属于 JSON.org 网站。

`json-ietf` 邮件列表
: 维护 JSON IETF 标准的 JSON IETF 工作组会使用该邮件列表来进行沟通。

JSONauts
: 另一个优秀站点，包含 JSON 教程、工具和文章。

JSON Schema 标准工作组
: JSON Schema 标准在 GitHub 上进行维护。

Google 群组：JSON Schema
: 该 Google 群组与 JSON Schema 标准工作组相关联。

Google 群组：`api-craft`
: 该群组关注 API 的设计与开发。

关于作者

Tom Marrs 热衷于展示技术在商业上的价值。作为 TEKsystems 全球服务部门的企业架构师，他促使公司采用了多项新的 API 架构与技术——REST、微服务和 JSON。Tom 领导过各种企业级的 API、Web、移动端、云和 SOA 项目。作为敏捷开发的拥趸，Tom 获得了 Scrum 联盟的 CSM 认证，他也乐于对项目团队进行相关的辅导与训练。

除本书外，Tom 还为 DZone 编写过（2013 年年度下载量第一的）JSON 核心参考卡片。Tom 与其他人合作出版过 *JBoss at Work*（O'Reilly 出版社）；他还在以下会议上发表过演讲：O'Reilly Open Source Convention（OSCON）、No Fluff Just Stuff（NFJS）和 Great Indian Developer Summit（GIDS）。Tom 希望不久的将来能够继续在这些会议上进行分享。

关于封面

本书封面上的动物是一只西伯利亚松鸦（Perisoreus infaustus），这是一种栖息于欧亚大陆北部的小型鸟类。它们的栖息范围极其广阔：西至瑞典，东至中国。它们会在北方密集的针叶林中筑巢。

西伯利亚松鸦最长可达 29 厘米，重至 79 克。它们有着长长的尾巴和棕灰色的外表。这种鸟类是杂食性的，以浆果和种子为食，同时也会摄入昆虫、腐肉以及小型啮齿类动物。雌鸟会在每年的三四月间产卵，并在冬季来临前抚育幼仔。

证据表明，由于人类对森林的砍伐，欧洲地区的西伯利亚松鸦的数量正在日益减少。不过，因为该物种在亚洲地区广泛而又稀疏地分布，所以西伯利亚松鸦目前尚未列入濒危动物的名单中。

O'Reilly 封面上的许多动物都已濒临灭绝，但它们的存在对世界至关重要。想要了解如何帮助它们，可以登录 animals.oreilly.com。

本书封面的图片来自 *Riverside Natural History* 一书。



微信连接



回复“JSON”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

JSON实战

JSON已经成为RESTful接口设计的事实标准，并在互联网数据交换领域日益受亲睐，是搭建优雅、高效应用程序的得力工具。

本书系统展示如何使用JSON工具和消息/文档设计来搭建企业级应用程序与服务，既包括JSON基础知识，又涵盖大量操作实践与使用案例，是全面掌握JSON强大功能的明智之选。

- **熟悉JSON基础知识并学习如何对JSON数据进行建模**
- **学习如何在Node.js、Ruby on Rails以及Java中使用JSON**
- **使用JSON Schema构建JSON文档来设计并测试API**
- **使用JSON搜索工具来搜索JSON文档的内容**
- **使用JSON转换工具将JSON文档转换成其他数据格式**
- **比较HAL和jsonapi等JSON超媒体格式**
- **使用MongoDB来存储和处理JSON文档**
- **使用Apache Kafka在服务间交换JSON消息**

汤姆·马尔斯（Tom Marrs），拥有多年企业架构经验，领导过各种企业级的API、Web、移动端、云和SOA项目。目前任TEKsystems全球服务部门企业架构师，促使公司采用了包括REST、微服务和JSON在内的多项API架构与技术。Tom还是敏捷开发的拥趸，并获得Scrum联盟的CSM认证。

封面设计：Randy Comer 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／程序设计

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



ISBN 978-7-115-48555-7
定价：89.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks